


# 1 NumLin: Linear Types for Linear Algebra

2 Dhruv C. Makwana 

3 Unaffiliated [dhruvmakwana.com](http://dhruvmakwana.com)

4 [dcm41@cam.ac.uk](mailto:dcm41@cam.ac.uk)

5 Neelakantan R. Krishnaswami 

6 Department of Computer Science and Technology, University of Cambridge, United Kingdom

7 [nk480@cl.cam.ac.uk](mailto:nk480@cl.cam.ac.uk)

## 8 — Abstract —

9 We present NUMLIN, a functional programming language whose type system is designed to enforce  
10 the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). We do  
11 so through a brief description of its key features and several illustrative examples. We show that  
12 NUMLIN's type system is sound and that its implementation improves upon naïve implementations  
13 of linear algebra programs, almost towards C-levels of performance. By doing so, we demonstrate (a)  
14 that linear types are well-suited to expressing the APIs of low-level linear algebra libraries accurately  
15 and concisely and (b) that, despite the complexity of prior work on it, fractional permissions  
16 can actually be implemented using simple, well-known techniques and be used practically in real  
17 programs.

18 **2012 ACM Subject Classification** Theory of computation → Program specifications

19 **Keywords and phrases** numerical, linear, algebra, types, permissions, OCaml

20 **Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

21 **Supplement Material** [www.github.com/dc-mak/lt4la](http://www.github.com/dc-mak/lt4la)

## 22 1 Introduction

23 Programmers writing numerical software often find themselves caught on the horns of a  
24 dilemma. The foundational, low-level linear algebra libraries such as BLAS and LAPACK  
25 offer programmers very precise control over the memory lifetime and usage of vector and  
26 matrix values. However, this power comes paired with the responsibility to manually manage  
27 the memory associated with each array object, and in addition to bringing in the familiar  
28 difficulties of reasoning about lifetimes, aliasing and sharing that plague low-level systems  
29 programming; this also moves the APIs away from the linear-algebraic, mathematical style  
30 of thinking that numerical programmers want to use.

31 As a result, programmers often turn to higher-level languages such as Matlab, R and  
32 Numpy, which offer very high-level array abstractions that can be viewed as ordinary  
33 mathematical values. This makes programming safer, as well as making prototyping and  
34 verification much easier, since it lets programmers write programs which bear a closer  
35 resemblance to the formulas that the mathematicians and statisticians designing these  
36 algorithms prefer to work with, and ensures that program bugs will reflect incorrectly-  
37 computed values rather than heap corruption.

38 The intention is that these languages can use libraries BLAS and LAPACK, without  
39 having to expose programmers to explicit memory management. However, this benefit comes  
40 at a price: because user programs do not worry about aliasing, the language implementations  
41 cannot in general exploit the underlying features of the low-level libraries that let them  
42 explicitly manage and reuse memory. As a result, programs written in high-level statistical  
43 languages can be much less memory-efficient than programs that make full use of the powers  
44 the low-level APIs offer.



© Dhruv C. Makwana and Neelakantan R. Krishnaswami;  
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:44

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 So in practice, programmers face a tradeoff: they can eschew safety and exploit the full  
 46 power of the underlying linear algebra libraries, or they can obtain safety at the price of  
 47 unneeded copies and worse memory efficiency. In this work, show that this tradeoff is not a  
 48 fundamental one.

49 NUMLIN is a functional programming language whose type system is designed to enforce  
 50 the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). It  
 51 does so by combining linear types, fractional permissions, runtime errors and recursion into  
 52 a small, easily understandable, yet expressive set of core constructs.

53 NUMLIN allows a novice to understand and work with complicated linear algebra library  
 54 APIs, as well as point out subtle aliasing bugs and reduce memory usage in existing programs.  
 55 In fact, we were able to use NUMLIN to find linearity and aliasing bugs in a linear algebra  
 56 algorithm that was *generated* by another program *specifically designed to translate matrix*  
 57 *expressions into an efficient sequence of calls to linear algebra routines*. We were also able to  
 58 reduce the number of temporaries used by the same algorithm, using NUMLIN’s type system  
 59 to guide us.

60 NUMLIN’s implementation supports several syntactic conveniences as well as a *usable*  
 61 integration with real OCaml libraries.

## 62 1.1 Contributions

63 In this paper

- 64 ■ we describe NUMLIN, a linearly typed language for linear algebra programs
- 65 ■ we illustrate that NUMLIN’s design and features are well-suited to its intended domain  
 66 with progressively sophisticated examples
- 67 ■ we prove NUMLIN’s soundness, using a step-indexed logical relation
- 68 ■ we describe a very simple, unification based type-inference algorithm for polymorphic  
 69 fractional permissions (similar to ones used for parametric polymorphism), demonstrating  
 70 an alternative approach to dataflow analysis [5]
- 71 ■ we describe an implementation that is both compatible with and usable from existing  
 72 code
- 73 ■ we show an example of how using NUMLIN helped highlight linearity and aliasing bugs,  
 74 and reduce the memory usage of a *generated* linear algebra program
- 75 ■ we show that using NUMLIN, we can achieve parity with C for linear algebra routines,  
 76 whilst having much better static guarantees about the linearity and aliasing behaviour of  
 77 our programs.

## 78 2 NumLin Overview and Examples

### 79 2.1 Overview

80 The core type theory of NUMLIN is a nearly off-the-shelf linear type theory, supporting  
 81 familiar features such as linear function spaces  $A \multimap B$  and tensor products  $A \otimes B$ . We  
 82 adopt linearity – the restriction that each program variable be used at most once – since it  
 83 allows us to express purely functional APIs for numerical library routines that mutate arrays  
 84 and matrices [17]. Due to linearity, values cannot alias and are only used once, which means  
 85 that linearly-typed updates result in no *observable* mutation.

86 As a result, programmers can reason about NUMLIN expressions as if they were ordinary  
 87 mathematical expressions – as indeed they are! We are merely adopting a stricter type  
 88 discipline than usual to make managing memory safe.

### 2.1.1 Intuitionism: ! and Many

However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them *intuitionistically*, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding Many constructor and let Many <id> = .. in .. eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NUMLIN enforces simple syntactic restrictions on which values can be wrapped up in a Many constructor (details in Section 3).

### 2.1.2 Fractional Permissions

There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine `gemm`, which (rather tersely) stands for *GEneric Matrix Multiplication*. A *simplified* definition of `gemm`( $\alpha$ , A, B,  $\beta$ , C) is  $C := \alpha AB + \beta C$ . In this case, A and B may alias each other but neither may alias C, because it is being written to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a freed value is *out of scope* for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing *without* losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are *parameterised* by a *fraction*. A fraction is either 1 ( $2^0$ ) or exactly *half* of another fraction ( $2^{-k}$ , for natural  $k$ ). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a *borrow*: the programmer may read from the value but not write to or free it. Creating an array/matrix gives you ownership of it, so too does having one (with a fractional permission of  $2^0$ ) passed in as an argument.

In NUMLIN, we can produce two aliases of a single array/matrix, by *sharing* it. If the original alias had a fractional permission of  $2^{-k}$  then the two new aliases of it will have a fractional permission of  $2^{-(k+1)}$  each. Thanks to linearity, the original array/matrix with a fractional permission of  $2^{-k}$  will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of `free` and `set` (for mutation) require a *whole* ( $2^0$ ) permission.

If we have two aliases *to the same matrix* with *identical* fractional permissions ( $2^{-(k+1)}$ ), we can recombine or *unshare* them back into a single one, with a larger  $2^{-k}$  permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

### 2.1.3 Runtime Errors

Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that can fail at runtime (the other being dimension checks, such as for `gemm`). The check being performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

## 23:4 NumLin: Linear Types for Linear Algebra

```
let rec factorial ( !x : !int ) : !int =  
  if x < 0 || x = 0 then  
    1  
  else  
    x * factorial (x - 1) in factorial  
;;
```

■ **Figure 1** Factorial function in NUMLIN.

### 131 2.1.4 Recursion

132 The final feature of NUMLIN which makes it sufficiently expressive is recursion (and of  
133 course, conditional branches to ensure termination). Conditional branches are implemented  
134 by ensuring that both branches use the same set of linear values. A function can be recursive  
135 if it captures no linear values from its environment. Like with **Many**, this is enforced via  
136 simple syntactic restrictions on the definition of recursive functions.

## 137 2.2 Examples

### 138 2.2.1 Factorial

139 Although a factorial function (Figure 1) may seem like an aggressively pedestrian first example,  
140 in a linearly typed language such as NUMLIN it represents the culmination of many features.

141 To simplify the design and implementation of NUMLIN’s type system, recursive functions  
142 must have full type annotations (non-recursive functions need only their argument types  
143 annotated). Its body is a closed expression (with respect to the function’s arguments), so it  
144 type-checks (since it does not capture any linear values from its environment).

145 The only argument is `!x : !int`. The `!` annotation on `x` is a syntactic convenience for  
146 declaring the value to be used intuitionistically, its full and precise meaning is described in  
147 Section 4.1.1.

148 The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it  
149 does not). Any linear values used by the condition would not be in scope in either branch of  
150 the `if`-expression. Both branches use `x` differently: one ignores it completely and the other  
151 uses it twice.

152 All numeric and boolean literals are implicitly wrapped in a **Many** and all primitives  
153 involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically  
154 64-bit floating-point numbers). The short-circuiting `||` behaves in exactly the same way as a  
155 boolean-valued `if`-expression.

### 156 2.2.2 Summing over an Array

157 Now we can add fractional permissions to the mix: Figure 2 shows a simple, tail-recursive  
158 implementation of summing all the elements in an array. There are many new features; first  
159 among them is `!x0 : !elt`, the type of array/matrix elements (64-bit floating point).

160 Second is `(’x) (row: ’x arr)` which is an array with a universally-quantified fractional  
161 permission. In particular, this means the body of the function cannot mutate or free the  
162 input array, only read from it. If the programmer did try to mutate or free `row`, then they  
163 would get a helpful error message (Figure 3).

164 Alongside taking a `row: ’x arr`, the function also returns an array with exactly the  
165 same fractional permission as the `row` (which can only be `row`). This is necessary because of  
166 linearity: for the caller, the original array passed in as an argument would be out of scope

```

let rec sum_array (!i : !int) (!n : !int) (!x0 : !elt)
  ('x) (row : 'x arr) : 'x arr * !elt =
  if i = n then
    (row, x0)
  else
    let (row, !x1) = row[i] in
    sum_array (i + 1) n (x0 +. x1) 'x row in
    sum_array
;;

```

■ **Figure 2** Summing over an array in NUMLIN.

```

let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality: *)
(*      z arr *)
(* with *)
(*      'x arr *)
(*)
(* Var 'x is universally quantified *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In examples/sum_array.lt, at line: 7 and column: 19 *)

```

■ **Figure 3** Attempting to write to or free a read only array in NUMLIN.

167 for the rest of the expression, so it needs to be returned and then rebound to be used for the  
 168 rest of the function.

169 An example of this consuming and re-binding is in `let (row, !x1) = row[i]`. Indexing  
 170 is implemented as a primitive `get: 'x. 'x arr --o !int --o 'x arr * !elt`. Although  
 171 fractional permissions can be passed around explicitly (as done in the recursive call), they  
 172 can also be *automatically inferred at call sites*: `row[i] == get _ row i` takes advantage of  
 173 this convenience.

### 174 2.2.3 One-dimensional Convolution

175 Figure 4 extends the set of features demonstrated by the previous examples by mutating one  
 176 of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel  
 177 (array of weights) and an input vector. It modifies the input vector *in-place* by replacing  
 178 each `write[i]` with a weighted (as per the values in the kernel) sum of it and its neighbours;  
 179 intuitively, sliding a dot-product with the kernel across the vector.

180 What's implemented in Figure 4 is a *simplified* version of this idea, so as to not distract  
 181 from the features of NUMLIN. The simplifications are:

- 182 ■ the kernel has a length 3, so only the value of `write[i-1]` (prior to modification in the
- 183 previous iteration) needs to be carried forward using `x0`
- 184 ■ `write` is assumed to have length `n+1`
- 185 ■ `i`'s initial value is assumed to be 1
- 186 ■ `x0`'s initial value is assumed to be `write[0]`
- 187 ■ the first and last values of `write` are ignored.

188 Mutating an array is implemented similarly to indexing one: a primitive `set: z arr`  
 189 `--o !int --o !elt --o z arr`. It consumes the original array and returns a new array  
 190 with the updated value. `let written = write[i] := <exp>` is just syntactic sugar for `let`  
 191 `written = set write i <exp>`.

```

let rec simp_oned_conv
  (!i : !int) (!n : !int) (!x0 : !elt)
  (write : z arr) ('x) (weights : 'x arr)
  : 'x arr * z arr =
  if n = i then (weights, write) else
  let !w0 <- weights[0] in
  let !w1 <- weights[1] in
  let !w2 <- weights[2] in
  let !x1 <- write[i] in
  let !x2 <- write[i + 1] in
  let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
  simp_oned_conv (i + 1) n x1 written _ weights in
simp_oned_conv
;;

```

■ **Figure 4** *Simplified* one-dimensional convolution.

```

let !square ('x) (x : 'x mat) =
  let (x, (!m, !n)) = sizeM _ x in
  let (x1, x2) = shareM _ x in
  let answer <- new (m, n) [| x1 * x2 |] in
  let x = unshareM _ x1 x2 in
  (x, answer) in
square
;;

```

■ **Figure 5** Linear regression (OLS):  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

192 Since `write: z arr` (where `z` stands for  $k = 0$ , representing a fractional permission  
 193 of  $2^{-k} = 2^{-0} = 1$ ), we may mutate it, but since we only need to read from `weights`, its  
 194 fractional permission index can be universally-quantified. In the recursive call, we see `_` being  
 195 used explicitly to tell the compiler to *infer* the correct fractional permission based on the  
 196 given arguments.

## 197 2.2.4 Squaring a Matrix

198 *The most pertinent aspect of NUMLIN is the types of its primitives.* While the types of  
 199 operations such as `get` and `set` might be borderline obvious, the types of BLAS/LAPACK  
 200 routines become an *incredibly useful, automated check for using the API correctly.*

201 Figure 5 shows how a linearly-typed matrix squaring function may be written in NUMLIN.  
 202 It is a *non-recursive* function declaration (the return type is inferred). Since we would like  
 203 to be able to use a function like `square` more than once, it is marked with a `!` annotation  
 204 (which also ensures it captures no linear values from the surrounding environment).

205 To square a matrix, first, we extract the dimensions of the argument `x`. Then, because  
 206 we need to use `x` twice (so that we can multiply it by itself) but linearity only allows one use,  
 207 we use `shareM: 'x. 'x mat --o 'x s mat * 'x s mat` to split the permission `'x` (which  
 208 represents  $2^{-x}$ ) into two halves (`'x s`, which represents  $2^{-(x+1)}$ ).

209 Even if `x` had type `z mat`, sharing it now enforces the assumption of all BLAS/LAPACK  
 210 routines that any matrix which is written to (which, in NUMLIN, is always of type `z mat`)  
 211 does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating  
 212 way, the expression would not type check, and we would get an error similar to the one in  
 213 Figure 3.

```

let !lin_reg ('x) (x : 'x mat)
    ('y) (y : 'y mat) =
  let (x, (!_n, !m)) = sizeM _ x in
  let xy <- new (m, 1) [| xT * y |] in
  let x_T_x <- new (m, m) [| xT * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg
;;

```

■ **Figure 6** Linear regression (OLS):  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

214 The line `let answer <- new (m,n) [| x1 * x2 |]` is syntactic sugar for first creating  
 215 a new  $m \times n$  matrix (`let answer = matrix m n`) and then storing the result of the mul-  
 216 tiplication in it (`let ((x1, x2), answer) = gemm 1. _ (x1, false) _ (x2, false) 0.`  
 217 `answer`). `false` means the matrix should not be accessed with indices transposed.

218 By using some simple pattern-matching and syntactic sugar, we can:

- 219 ■ write normal-looking, apparently non-linear code
- 220 ■ use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK
- 221 routine inserted with appropriate re-bindings
- 222 ■ retain the safety of linear types with fractional permissions by having the compiler
- 223 statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK
- 224 routines.

## 225 2.2.5 Linear Regression

226 In Figure 6, we wish to compute  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ . To do that, first, we extract the  
 227 dimensions of matrix  $\mathbf{x}$ . Then, we say we would like  $\mathbf{xy}$  to be a new matrix, of dimension  
 228  $m \times 1$ , which contains the result of  $\mathbf{X}^T \mathbf{y}$  (using syntactic sugar for `matrix` and `gemm` calls  
 229 similar to that used in Figure 5, with a <sup>T</sup> annotation on  $\mathbf{x}$  to set  $\mathbf{x}$ 's 'transpose indices'-flag  
 230 to `true`).

231 However, the line `let x_T_x <- new (m,m) [| xT * x |]`, works for a slightly differ-  
 232 ent reason: that pattern is matched to a BLAS call to (`syrk true 1. x 0. x_T_x`), which  
 233 only uses  $\mathbf{x}$  once. Hence  $\mathbf{x}$  can appear *twice* in the *pattern* without any calls to `share`.

234 After computing  $\mathbf{x}_T \mathbf{x}$ , we need to invert it and then multiply it by  $\mathbf{xy}$ . The BLAS  
 235 routine `posv: z mat --o z mat --o z mat * z mat` does exactly that: assuming the first  
 236 argument is symmetric, `posv` mutates its second argument to contain the desired value. Its  
 237 first argument is also mutated to contain the (upper triangular) Cholesky decomposition  
 238 factor of the original matrix. Since we do not need that matrix (or its memory) again, we  
 239 `free` it. If we forgot to, we would get a `Variable to_del not used` error. Lastly, we return  
 240 the `answer` alongside the untouched input matrices  $(\mathbf{x}, \mathbf{y})$ .

## 241 2.2.6 L1-Norm Minimisation on Manifolds

242 L1-Norm minimisation is often used in optimisation problems, as a *regularisation* term for  
 243 reducing the influence of outliers. Although the below formulation[8] is intended to be used  
 244 with *sparse* computations, NUMLIN's current implementation only implements dense ones.  
 245 However, it still serves as a useful example of explaining NUMLIN's features.



```

let !l1_norm_min (q : z mat) (u : z mat) =
  let (u, (!_n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n, q_inv_u) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min
;;

```

■ **Figure 7** L1-norm minimisation on manifolds:  $\mathbf{Q}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{U}^T\mathbf{Q}^{-1}\mathbf{U})^{-1}\mathbf{U}^T$

Figure 7 shows even more pattern-matching. Patterns of the form `let <id> <- [| beta * c + alpha * a * b |]` are also desugared to `gemm` calls. Primitives like `transpose`: `'x. 'x mat --o 'x mat * z mat` and `eye`: `!int --o z mat` allocate new matrices; `transpose` returns the transpose of a given matrix and `eye k` evaluates to a  $k \times k$  identity matrix.

We also see our first example of re-using memory for different matrices: like with `to_del` and `posv` in the previous example, we do not need the value stored in `tmp_5_5` after the call to `gesv` (a primitive similar to `posv` but for a non-symmetric first argument). However, we can re-use its memory much later to store `answer` with `let answer <- [| 0. * tmp_5_5 + q_inv_u * inv_u_T |]`. Again, thanks to linearity, the identifiers `q` and `tmp_5_5` are out of scope by the time `answer` is bound. Although during execution, all three refer to the same piece of memory, logically they represent different values throughout the computation.

## 2.2.7 Kalman Filter

A *Kalman Filter*[12] is an algorithm for combining prior knowledge of a state, a statistical model and measurements from (noisy) sensors to produce an estimate a more reliable estimated of the current state. It has various applications (navigation, signal-processing, econometrics) and is relevant here because it is usually presented as a series of complex matrix equations.

Figure 8 shows a NUMLIN implementation of a Kalman filter (equations in Figure 9). A few new features and techniques are used in this implementation:

- `sym` annotations in matrix expressions: when this is used, a call to `symm` (the equivalent of `gemm` but for symmetric matrices so that only half the operations are performed) is inserted
- `copyM_to` is used to re-use memory by *overwriting* the contents of its second argument to that of its first (erroring if dimensions do not match)
- `let new_r <- new [| r_2 |]` creates a copy of `r_2`
- `posvFlip` is like `posv` except for solving  $XA = B$
- a lot of memory re-use; the following sets of identifiers alias each other:
  - `r_1`, `r_2` and `k_by_k`
  - `data_1` and `data_2`
  - `mu` and `new_mu`
  - `sigma_hT` and `x`



```

let !kalman
  ('s) (sigma : 's mat) (* n,n *)
  ('h) (h : 'h mat)      (* k,n *)
  (mu : z mat)            (* n,1 *)
  (r_1 : z mat)           (* k,k *)
  (data_1 : z mat)        (* k,1 *) =
  let (h, (!k, !n)) = sizeM _ h in
  (* could use [| sym(sigma) * hT |] but would
     need a (n,k) temporary hT = transpose _ h *)
  let sigma_hT <- new (n, k) [| sigma * h^T |] in
  let r_2 <- [| r_1 + h * sigma_hT |] in
  let (k_by_k, x) = posvFlip r_2 sigma_hT in
  let data_2 <- [| h * mu - data_1 |] in
  let new_mu <- [| mu + x * data_2 |] in
  let x_h <- new (n,n) [| x * h |] in
  let () = freeM (* n,k *) x in
  let sigma2 <- new [| sigma |] in
  let new_sigma <- [| sigma2 - x_h * sym(sigma) |] in
  let () = freeM (* n,n *) x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2)))) in
kalman
;;

```

■ **Figure 8** Kalman filter: see Figure 9 for the equations this code implements and Figure 17 for an equivalent CBLAS/LAPACK implementation.

$$\begin{aligned}\mu' &= \mu + \Sigma H^T (R + H \Sigma H^T)^{-1} (H \mu - \text{data}) \\ \Sigma' &= \Sigma (I - H^T (R + H \Sigma H^T)^{-1} H \Sigma)\end{aligned}$$

■ **Figure 9** Kalman filter equations (credit: matthewrocklin.com).

277 The NUMLIN implementation is much longer than the mathematical equations for two  
 278 reasons. First, the NUMLIN implementation is a let-normalised form of the Kalman equations:  
 279 since there a large number of unary/binary (and occasionally ternary) sub-expressions in  
 280 the equations, naming each one line at a time makes the implementation much longer.  
 281 Second, NUMLIN has the additional task of handling explicit allocations, aliasing and frees  
 282 of matrices. However, it is exactly this which makes it possible (and often, easy) to spot  
 283 additional opportunities for memory re-use. Furthermore, a programmer can explore those  
 284 opportunities easily because NUMLIN's type system statically enforces correct memory  
 285 management and the aliasing assumptions of BLAS/LAPACK routines.

## 286 3 Formal System

### 287 3.1 Core Type Theory

288 The full typing rules are in Appendix A.1, but the key ideas are as follow.

- 289 ■ A typing judgement consists of  $\Theta; \Delta; \Gamma \vdash e : t$ .
- 290 ■  $\Theta$  is the environment that tracks which fractional permission variables in scope. Fractional  
 291 permissions (the Perm judgement) and types (the Type judgement) are *well-formed* if all  
 292 of their free fractional variables are in  $\Theta$ .
- 293 ■  $\Delta$  is the environment storing non-linearly or *intuitionistically* typed variables.

## 23:10 NumLin: Linear Types for Linear Algebra

294 ■  $\Gamma$  is the environment storing linearly typed variables.

295 Note that rules for typing  $()$ , booleans, integers and elements are typed with respect to  
 296 an *empty* linear environment: this means no linear values are needed to produce a value of  
 297 those types.

298 
$$\frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \quad \text{TY\_UNIT\_INTRO}$$

299 Conversely, whenever two or more subexpressions need to be typed, they must consume  
 300 a disjoint set of linear values (pairs, let-expressions). In the case of if-expressions, both  
 301 branches must consume the same set of linear values (disjoint to the ones used to evaluate  
 302 the condition).

303 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !\mathbf{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if } e \mathbf{ then } e_1 \mathbf{ else } e_2 : t} \quad \text{TY\_BOOL\_ELIM}$$

304 The **Many** introduction and elimination rules are very important. Producing  $!$ -type values  
 305 may only be done if the expression inside is a syntactic value which is not a location. This  
 306 allows all safely duplicable resources, including functions which capture non-linear resources  
 307 from their environments, but prevents producing aliases of (pointers to) arrays and matrices.  
 308 This is exactly the same as value-restriction from the world of parametric polymorphism.

309 
$$\frac{\begin{array}{l} \Theta; \Delta; \cdot \vdash v : t \\ v \neq l \end{array}}{\Theta; \Delta; \cdot \vdash \mathbf{Many } v : !t} \quad \text{TY\_BANG\_INTRO}$$

310 Consuming a  $!$ -type value *moves it* from the linear environment  $\Gamma$  and *into* the intu-  
 311 itionistic environment  $\Delta$ . This is exactly why  $\mathbf{let } !x = e_1 \mathbf{ in } e_2$  desugars to  $\mathbf{let } \mathbf{Many } x =$   
 312  $e_1 \mathbf{ in let } \mathbf{Many } x = \mathbf{Many } (\mathbf{Many } x) \mathbf{ in } e_2$ .

313 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let } \mathbf{Many } x = e \mathbf{ in } e' : t'} \quad \text{TY\_BANG\_ELIM}$$

314 Rules TY\_GEN and TY\_SPC are for fractional permission generalisation and specialisa-  
 315 tion respectively. They allow the definition and use of functions that are polymorphic in the  
 316 fractional permission index of their results and one or more of their arguments.

317 
$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun } fc \rightarrow e : \forall fc. t} \quad \text{TY\_GEN} \qquad \frac{\begin{array}{l} \Theta \vdash f \text{ Perm} \\ \Theta; \Delta; \Gamma \vdash e : \forall fc. t \end{array}}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \quad \text{TY\_SPC}$$

318 Rule TY\_FIX shows how recursive functions are typed. Even though recursive functions  
 319 are fully annotated, type checking them is interesting for two reasons: to type check the  
 320 body of the fixpoint, the type of the recursive function is in the *intuitionistic* environment  $\Delta$   
 321 (without this, you would not be able to write a base case) whilst the argument and its type

are the *only things in the linear environment*  $\Gamma$ . The latter means that recursive functions can be type checked in an empty environment (thus be wrapped in **Many** and used zero or multiple times).

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix}(g, x : t, e : t') : t \multimap t'} \quad \text{TY\_FIX}$$

Lastly, types of almost all NUMLIN primitives, as embedded in OCaml's type system, are shown in Appendix G, with some similar ones (like those for binary arithmetic operators) omitted for brevity. The main difference between the OCaml type of a primitive like **gemm** and its NUMLIN counterpart is the inclusion of explicit ' $\forall$ 's. So, **float bang**  $\rightarrow$  (**'a mat** \* **bool bang**)  $\rightarrow$  (**'b mat** \* **bool bang**)  $\rightarrow$  **float bang**  $\rightarrow$  **z mat**  $\rightarrow$  (**'a mat** \* **'b mat**) \* **z mat** will correspond to

$$\mathbf{!elt} \multimap \forall x. x \mathbf{mat} \otimes \mathbf{!bool} \multimap \forall y. y \mathbf{mat} \otimes \mathbf{!bool} \multimap \mathbf{!elt} \multimap z \mathbf{mat} \multimap (x \mathbf{mat} \otimes y \mathbf{mat}) \otimes z \mathbf{mat}$$

### 3.2 Dynamic Semantics

The full, small-step transition relation is in Appendix A.2, but the key ideas are as follow.

Heaps ( $\sigma$ ) are multisets containing triples of an abstract location  $l$ , a fractional permission  $f$  and sized matrices  $m_{n,k}$ . The notation  $l \mapsto_f m_{k_1, k_2}$  should be read as "location  $l$  represents  $f$  ownership over matrix  $m$  (of size  $k_1 \times k_2$ )". Each heap-and-expression either steps to another heap-and-expression or a runtime error **err**. In the full grammar definition we see a definition of values and contexts in the language.

We draw the reader's attention to the definitions relating to fractional permissions. Specifically, unlike a lambda, the body of a **fun**  $fc \rightarrow \_$  must be a syntactic value. The context **fun**  $fc \rightarrow [-]$  means expressions can be reduced inside a fractional permission generalisation. This is to emphasize that fractions are merely *compile-time constructs* and do not affect runtime behaviour. Correct usage of fractions is enforced by the type system, so programs do not get stuck. Fractional permissions are specialised using substitution over both the heap and an expression (OP\_FRAC\_PERM).

$$\frac{}{\langle \sigma, (\mathbf{fun} fc \rightarrow v)[f] \rangle \rightarrow \langle \sigma[f c / f], v[f c / f] \rangle} \quad \text{OP\_FRAC\_PERM}$$

Like with the static semantics, the interesting rules in the dynamic semantics are those relating to primitives. Creating a matrix (**matrix**  $k_1 k_2$ ) successfully (OP\_MATRIX) requires non-negative dimensions and returns a (fresh) location of a matrix of those dimensions, extending the heap to reflect that  $l$  represents a complete ownership over the new matrix.

$$\frac{0 \leq k_1, k_2 \quad l \text{ fresh}}{\langle \sigma, \mathbf{matrix} k_1 k_2 \rangle \rightarrow \langle \sigma + \{l \mapsto_1 M_{k_1, k_2}\}, l \rangle} \quad \text{OP\_MATRIX}$$

Dually, OP\_FREE, requires a location represent complete ownership before removing it and the matrix it points to from the heap.

$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1, k_2}\}, \mathbf{free} l \rangle \rightarrow \langle \sigma, () \rangle} \quad \text{OP\_FREE}$$

Choosing a multiset representation as opposed to a set allows for two convenient invariants: multiplicity of a triple  $l \mapsto_f m_{k_1, k_2}$  in the heap corresponds to the number of aliases of  $l$  in the

## 23:12 NumLin: Linear Types for Linear Algebra

expression with type  $f$  **mat** and the sum of all the fractions for  $l$  will always be 1 (for a closed, well-typed expression). With this in mind, the rules `OP_SHARE` and `OP_UNSHARE_EQ` are fairly natural.

$$\frac{\langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, \mathbf{share}[f] l \rangle \rightarrow \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, (l, l) \rangle}{\text{OP\_SHARE}}$$

$$\frac{\langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, \mathbf{unshare}[f] l l \rangle \rightarrow \langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, l \rangle}{\text{OP\_UNSHARE\_EQ}}$$

Combining all of these features, we see that `OP_GEMM_MATCH` requires that the location being updated ( $l_3$ ) has complete ownership of over matrix  $m_3$  and can thus change what value it stores to  $m_1 m_2 + m_3$ . In particular, this places no restriction on  $l_2$  and  $l_3$ : they could be **shared** aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure:  $\mapsto_1$  for any locations that are written to and  $\mapsto_{fc}$  for anything else.

$$\frac{\begin{aligned} \sigma' &\equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1k_1, k_2}\} + \{l_2 \mapsto_{fc_2} m_{2k_2, k_3}\} \\ \sigma_1 &\equiv \sigma' + \{l_3 \mapsto_1 m_{3k_1, k_3}\} \\ \sigma_2 &\equiv \sigma' + \{l_3 \mapsto_1 (m_1 m_2 + m_3)_{k_1, k_3}\} \end{aligned}}{\langle \sigma_1, \mathbf{gemm}[fc_1] l_1 [fc_2] l_2 l_3 \rangle \rightarrow \langle \sigma_2, ((l_1, l_2), l_3) \rangle} \text{OP\_GEMM\_MATCH}$$

### 3.3 Logical Relation

First, we define an interpretation of heaps with fractional permissions in the style of Bornat et. al [6] (interpreting the multiset as a partial map from locations to the sum of all its associated fractions and a matrix) as well as the  $n$ -fold iteration of  $\rightarrow$ .

$$\mathcal{H}[\sigma] = \star_{(l, f, m) \in \sigma} [l \mapsto_f m]$$

where

$$(\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \wedge l \notin \text{dom}(\varsigma_2) \\ \varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \wedge l \notin \text{dom}(\varsigma_1) \\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \wedge (f_2, m) = \varsigma_2(l) \wedge f_1 + f_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We then define a step-indexed logical relation in the style of Morrisett et. al [14].  $(\varsigma, v) \in \mathcal{V}_k[t]$  means it takes a heap with exactly  $\varsigma$  resources to produce a value  $v$  of type  $t$  in at most  $k$  steps. So, something like a **unit** or a  $!t$  need no resources, whereas a  $f$  **mat** needs exactly  $f$  ownership of a some matrix and a pair needs a  $\star$  combination of the heaps required for each component.

$$\begin{aligned} \mathcal{V}_k[\mathbf{unit}] &= \{(\emptyset, *)\} \\ \mathcal{V}_k[f \mathbf{mat}] &= \{(\{l \mapsto_{2-f} \_ \}, l)\} \\ \mathcal{V}_k[!t] &= \{(\emptyset, \mathbf{Many} v) \mid (\emptyset, v) \in \mathcal{V}_k[t]\} \\ \mathcal{V}_k[t_1 \otimes t_2] &= \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[t_1] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[t_2]\} \end{aligned}$$

The definition of  $\mathcal{V}_k[\forall fc. t]$  says a value and heap must be the same regardless of what fraction is substituted into both; the  $k - 1$  is to take into account fraction specialisation takes ones step (`OP_SPC`).

$$\mathcal{V}_k[\forall fc. t] = \{(\varsigma, \mathbf{fun} fc \rightarrow v) \mid \forall f. (\varsigma[fc/f], v[fc/f]) \in \mathcal{V}_{k-1}[t[fc/f]]\}$$

393 To understand the definition of  $\mathcal{V}_k[t' \multimap t]$ , we must first look at  $\mathcal{C}_k[t]$ , the computational  
 394 interpretation of types. Intuitively, it is a combination of a frame rule on heaps (no  
 395 interference), type-preservation and termination (in  $j < k$  steps) to either an error or a  
 396 heap-and-expression, with the further condition that if the expression is a syntactic value  
 397 then it is also one semantically.

$$\begin{aligned} 398 \quad \mathcal{C}_k[t] = \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r. \varsigma_s \star \sigma_r \text{ defined} \Rightarrow \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \\ 399 \quad \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \sigma_r, e_f) \in \mathcal{V}_{k-j}[t])\} \end{aligned}$$

401 In this light,  $\mathcal{V}_k[t' \multimap t]$  simply says that  $v$  is a function and that evaluating the application  
 402 of it to any argument (of the correct type, requiring its own set of resources, bounded by  $k$   
 403 steps) satisfies all the aforementioned properties.

$$\begin{aligned} 404 \quad \mathcal{V}_k[t' \multimap t] = \{(\varsigma_v, v) \mid (v \equiv \mathbf{fun} \ x : t' \rightarrow e \vee v \equiv \mathbf{fix}(g, x : t', e : t)) \wedge \\ 405 \quad \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[t']. \varsigma_v \star \varsigma_{v'}' \text{ defined} \Rightarrow (\varsigma_v \star \varsigma_{v'}', v v') \in \mathcal{C}_j[t]\} \end{aligned}$$

407 The interpretation of typing environments  $\Delta$  and  $\Gamma$  are with respect to an arbitrary  
 408 substitution of fractional permissions  $\theta$ . Note that only the interpretation of  $\Gamma$  involves a  
 409 (potentially) non-empty heap.

$$\begin{aligned} 410 \quad \mathcal{I}_k[\Delta, x : t]\theta = \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\Delta]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\theta(t)]\} \\ 411 \quad \mathcal{L}_k[\Gamma, x : t]\theta = \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\theta(t)]\} \end{aligned}$$

413 And so the final semantic interpretation of a typing judgement simply quantifies over  
 414 all possible fractional permission substitutions  $\theta$ , linear value substitutions  $\gamma$ , intuitionistic  
 415 value substitutions  $\delta$  and heaps  $\sigma$ . Note that,  $\varsigma \equiv \mathcal{H}[\theta(\sigma)]$ .

$$\begin{aligned} 416 \quad {}_k[\Theta; \Delta; \Gamma \vdash e : t] = \forall \theta, \delta, \gamma, \sigma. \Theta = \text{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge \delta \in \mathcal{I}_k[\Delta]\theta \Rightarrow \\ 417 \quad (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)] \end{aligned}$$

### 419 3.4 Soundness Theorem

420 ► **Theorem 1.** (*The Fundamental Lemma of Logical Relations*)

$$421 \quad \forall \Theta, \Delta, \Gamma, e, t. \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k. {}_k[\Theta; \Delta; \Gamma \vdash e : t]$$

#### 422 3.4.1 Explanation

423 If an expression is syntactically type-checked, then for an arbitrary number of steps  $k$ , under  
 424 any substitution  $\theta$  of free fractional-permission variables, linear variables  $\gamma$  (with a suitable  
 425 heap  $\varsigma$ ) and intuitionistic variables  $\delta$ , that suitable heap and expression  $(\varsigma, \theta(\delta(\gamma(e))))$  are in  
 426 the computational interpretation of the type of that expression  $\mathcal{C}_k[\theta(t)]$ .

427 The *computational interpretation* is as defined before: a combination of a frame rule on  
 428 heaps (no interference), type-preservation and termination (in  $j < k$  steps) to either an error  
 429 or a heap-and-expression, with the further condition that if the expression is a syntactic  
 430 value then it is also one semantically.

431 It identifies executions that do no un- or ill-defined behaviours (e.g. adding a boolean and  
 432 an integer). Since our operational semantics explicitly models deallocation, we now know no  
 433 well-typed program will ever try to access deallocated memory, establishing the correctness  
 434 of our memory management checking.

### 435 3.4.2 Proof Sketch

436 To prove the above theorem, we need several lemmas; the interesting ones are: the moral  
 437 equivalent of the frame rule (C.1), monotonicity for the step-index (C.5), splitting up  
 438 environments corresponds to splitting up heaps (C.7) and heap-and-expressions take the  
 439 same steps of evaluation under any substitution of their free fractional permissions (C.8).

440 The proof proceeds by induction on the typing judgement. The case for `TY_FIX` is the  
 441 reason we quantify over the step-index  $k$  in the *conclusion* of the soundness theorem. It  
 442 allows us to then induct over the step-index and assume exactly the thing we need to prove  
 443 at a smaller index.

444 The case for `TY_GEN` follows a similar pattern, but has the extra complication of reducing  
 445 an expression with an arbitrary fractional permission variable in it, and then instantiating it  
 446 at the last moment to conclude, which is where C.8 (heap-and-expressions take the same  
 447 steps of evaluation under any substitution of their free fractional permissions) is used.

448 The rest of the cases are either very simple base cases (variables, unit, boolean, integer  
 449 or element literals) or follow very similar patterns; for these, only `TY_LET` is presented in  
 450 full and other similar cases simply highlight exactly what would be different. The general  
 451 idea is to split up the linear substitution and heap along the same split of  $\Gamma/\Gamma'$ , then (by  
 452 induction) use  $\mathcal{C}_k[-]$  and one ‘half’ of the linear substitution and heap to conclude the ‘first’  
 453 sub-expression either takes  $j < k$  steps to `err` or another heap-and-expression.

454 In the first case, you use `OP_CONTEXT_ERR` to conclude the whole let-expression does  
 455 the same. Similarly we use `OP_CONTEXT`  $j$  times in the second case. However, a small  
 456 book-keeping wrinkle needs to be taken care of in the case that the heap-and-expression  
 457 turns into a value in  $i \leq j$  steps: `OP_CONTEXT` is not functorial for the  $n$ -fold iteration of  
 458  $\rightarrow$ . Basically, the following is not quite true:

$$459 \frac{\langle \sigma, e \rangle \rightarrow^j \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow^j \langle \sigma', C[e'] \rangle} \text{ OP\_CONTEXT}$$

460 because after the  $i$  steps, we need to invoke `OP_LET_VAR` to proceed evaluation for any  
 461 remaining  $j - i$  steps. After that, it suffices to use the induction hypothesis on the second  
 462 sub-expression to finish the proof. To do so, we need to construct a valid linear substitution  
 463 and heap (i.e., one in  $\mathcal{L}_k[\Gamma', x : t]\theta$ ). We take the other ‘half’ of the linear substitution and  
 464 heap (from the initial split at the start) and extend it with  $[x \mapsto v]$ , (where  $x$  is the variable  
 465 bound in the let-expression and  $v$  is the value we assume the first sub-expression evaluated  
 466 to in  $i$  steps).

## 467 4 Implementation

### 468 4.1 Implementation Strategy

469 `NUMLIN` transpiles to OCaml and its implementation follows the structure of a typical  
 470 domain-specific language (DSL) compiler. Although `NUMLIN`’s current implementation is  
 471 not as an embedded DSL, its the general design is simple enough to adapt to being so and  
 472 also to target other languages.

473 Alongside the transpiler, a ‘Read-Check-Translate’ loop, benchmarking program and a  
 474 test suite are included in the artifacts accompanying this paper.

- 475 1. **Parsing.** A generated, LR(1) parser parses a text file into a syntax tree. In general, this  
 476 part will vary for different languages and can also be dealt with using combinators or  
 477 syntax-extensions (the EDSL approach) if the host language offers such support.

- 478 **2. Desugaring.** The syntax tree is then desugared into a smaller, more concise, abstract  
 479 syntax tree. This allows for the type checker to be simpler to specify and easier to  
 480 implement.
- 481 **3. Matrix Expressions** are also desugared into the abstract syntax tree through pattern-  
 482 matching.
- 483 **4. Type checking.** The abstract syntax tree is explicitly typed, with some inference to  
 484 make writing typical programs more convenient.
- 485 **5. Code Generation.** The abstract syntax tree is translated into OCaml, with a few  
 486 ‘optimisations’ to produce more readable code. This process is type-preserving: NUMLIN’s  
 487 type system is embedded into OCaml’s (Figure 11), so the OCaml type checker acts as a  
 488 sanity check on the generated code.

489 A very pleasant way to use NUMLIN is to have the build system generate code at *compile-*  
 490 *time* and then have the generated code be used by other modules like normal OCaml functions.  
 491 This makes it possible and even easy to use NUMLIN alongside existing OCaml libraries;  
 492 in fact, this is exactly how the benchmarking program and test-suite use code written in  
 493 NUMLIN.

#### 494 4.1.1 Desugaring, Matrix Expressions and Type Checking

495 Desugaring is conventional, outlined in Appendix F. Matrix expression are translated into  
 496 BLAS/ LAPACK calls via purely syntactic pattern-matching, outlined in Figure 10.

#### 497 4.1.2 Type checking

498 Type checking is mostly standard for a linearly typed language, with the exception of  
 499 fractional permission inference. By restricting fractions to be non-positive integer powers  
 500 of two, we only need to keep track of the logarithm of the fractions used. Explicit sharing  
 501 and unsharing removes the need for performing dataflow analysis. As a result, all fractional  
 502 arithmetic can be solved with unification, and in doing so, fractions become directly usable  
 503 in NUMLIN’s type-system as opposed to a convenient theoretical tool.

504 Because all functions must have their argument types explicitly annotated, inferring the  
 505 correct fraction at a call-site is simply a matter of unification. We believe *full-inference*  
 506 *of fractional permissions is similarly just matter of unification* (thanks to an experimental  
 507 implementation of just this feature), even though the formal system we present here is for an  
 508 explicitly-typed language.

509 There are a few differences between the type system as presented in 3.2 and how we  
 510 implemented it: the environment *changes* as a result of type checking an expression (the  
 511 standard transformation to avoid a non-deterministic split of the environment for checking  
 512 pairs); variables are *marked as used* rather than removed for better error messages; variables  
 513 are *tagged* as linear or intuitionistic in *one* environment as opposed to being stored in *two*  
 514 separate ones (this allows scoping/variable look-up to be handled uniformly).

#### 515 4.1.3 Code Generation

516 is a straightforward mapping from NUMLIN’s core constructs to high-level OCaml ones. We  
 517 embed NUMLIN’s type- and term- constructors into OCaml as a sanity check on the output  
 518 (Figure 11).

519 This is also useful when using NUMLIN from within OCaml; for example, we can use  
 520 existing tools to inspect the type of the function we are using (Figure 12). It is worth



$$\begin{aligned}
& \text{let } v \leftarrow x[e] \text{ in } e \Rightarrow \text{let } (x, !v) = x[e] \text{ in } e \quad (\text{similarly for matrices}) \\
& \text{let } x_2 \leftarrow \text{new } [|x_1|] \text{ in } e \Rightarrow \text{let } (x_1, x_2) = \text{copyM\_} x_1 \text{ in } e \\
& \text{let } x_2 \leftarrow [|x_1|] \text{ in } e \Rightarrow \text{let } (x_1, x_2) = \text{copyM\_to\_} x_1 x_2 \text{ in } e \\
\\
& M ::= X \mid X^T \mid \text{sym}(X) \\
\\
& \text{let } Y \leftarrow \text{new } (n, k) [| \alpha M_1 M_2 |] \text{ in } e \Rightarrow \\
& \quad \text{let } Y = \text{matrix } n \ k \text{ in let } Y \leftarrow [| \alpha M_1 M_2 + 0Y |] \text{ in } e \\
& \text{let } Y \leftarrow [| \alpha X X^T + \beta Y |] \text{ in } e \Rightarrow \\
& \quad \text{let } (X, Y) = \text{syrk false } \alpha \_ X \beta Y \text{ in } e \\
& \text{let } Y \leftarrow [| \alpha X^T X + \beta Y |] \text{ in } e \Rightarrow \\
& \quad \text{let } (X, Y) = \text{syrk true } \alpha \_ X \beta Y \text{ in } e \\
& \text{let } Y \leftarrow [| \alpha \text{sym}(X_1) X_2 + \beta Y |] \text{ in } e \Rightarrow \\
& \quad \text{let } ((X_1, X_2), Y) = \text{symm false } \alpha \_ X_1 \_ X_2 \beta Y \text{ in } e \\
& \text{let } Y \leftarrow [| \alpha X_2 \text{sym}(X_1) + \beta Y |] \text{ in } e \Rightarrow \\
& \quad \text{let } ((X_1, X_2), Y) = \text{symm true } \alpha \_ X_1 \_ X_2 \beta Y \text{ in } e \\
& \text{let } Y \leftarrow [| \alpha X_1^{T?} X_2^{T?} + \beta Y |] \text{ in } e \Rightarrow \\
& \quad \text{let } ((X_1, X_2), Y) = \text{gemm } \alpha \_ (X_1, \text{true}_{\text{false}}) \_ (X_2, \text{true}_{\text{false}}) \beta Y \text{ in } e
\end{aligned}$$

■ **Figure 10** Purely syntactic pattern-matching translations of matrix expressions.

reiterating that only the type- and term- constructors are translated into OCaml, NUMLIN’s precise control over linearity and aliasing are not brought over.

We actually use this fact to our advantage to clean up the output OCaml by removing what would otherwise be redundant re-bindings (Figure 13). Combined with a code-formatter, the resulting code is not obviously correct and exactly what an expert would intend to write by hand, but now with the guarantees and safety of NUMLIN behind it. A small example is shown in Figure 14, a larger one in Figure 16.

## 4.2 Performance Metrics

We think that using NUMLIN has two primary benefits: safety and performance. We discuss safety in 5.1, where we describe how we used NUMLIN to find linearity and aliasing bugs in a linear algebra algorithm that was *generated* by another program.

### 4.2.1 Setup

For performance, we measured the execution times of four equivalent implementations of a Kalman filter: in C (using CBLAS), NUMLIN (using OWL’s low-level CBLAS bindings), OCaml (using OWL’s intended, safe/copying-by-default interface), and Python (using NUMPY, with the interpreter started and functions interpreted). We measured execution time in microseconds, against an exponentially (powers of 5) increasing scaling factor for matrix size parameters  $n = 5$  and  $k = 3$ .

$f ::=$		
$fc$	<code>module Arr =</code>	$\llbracket fc \rrbracket = 'fc$
$Z$	<code>Owl.Dense.Ndarray.D</code>	$\llbracket Z \rrbracket = z$
$S f$	<code>type z = Z</code>	$\llbracket S f \rrbracket = \llbracket f \rrbracket s$
$t ::=$	<code>type 'a s = Succ</code>	$\llbracket \text{unit} \rrbracket = \text{unit}$
<code>unit</code>	<code>type 'a arr =</code>	$\llbracket \text{bool} \rrbracket = \text{bool}$
<code>bool</code>	<code>A of Arr.arr</code>	$\llbracket \text{int} \rrbracket = \text{int}$
<code>int</code>	<code>[@@unboxed]</code>	$\llbracket \text{elt} \rrbracket = \text{float}$
<code>elt</code>	<code>type 'a mat =</code>	$\llbracket f \text{ arr} \rrbracket = \llbracket f \rrbracket \text{ arr}$
<code>f arr</code>	<code>M of Arr.arr</code>	$\llbracket f \text{ mat} \rrbracket = \llbracket f \rrbracket \text{ mat}$
<code>f mat</code>	<code>[@@unboxed]</code>	$\llbracket ! t \rrbracket = \llbracket t \rrbracket \text{ bang}$
<code>! t</code>	<code>type 'a bang =</code>	$\llbracket \forall fc. t \rrbracket = \llbracket t \rrbracket$
$\forall fc. t$	<code>Many of 'a</code>	$\llbracket t \otimes t' \rrbracket = \llbracket t \rrbracket * \llbracket t' \rrbracket$
$t \otimes t'$	<code>[@@unboxed]</code>	$\llbracket t \multimap t' \rrbracket = \llbracket t \rrbracket \rightarrow \llbracket t' \rrbracket$
$t \multimap t'$		

■ **Figure 11** NUMLIN's type grammar (left) and its embedding into OCaml (right).

```

1 let lt4la_kalman ~sigma ~h ~mu ~r ~data =
0   Examples.Kalman.it (M sigma) (M h) (M mu) (M r) (M data)
NORMAL test/examples_test.ml
'a mat ->
'b mat ->
'c mat ->
z mat ->
z mat -> ('a mat * ('b mat * ('c mat * (z mat * z mat)))) * (z mat * z mat)
:merlin-type-history:
0   let fact = Examples.Factorial.it in
NORMAL test/examples_test.ml
int bang -> int bang

```

■ **Figure 12** Using NUMLIN functions from OCaml.

539 For large scaling factors ( $n = 5^4, 5^5$ ), we triggered a full garbage-collection before  
540 measuring the execution time of a single call of a function. However, due to the limitations  
541 of the micro-benchmarking library we used, for smaller scaling factors ( $n = 5^1, 5^2, 5^3$ ), we  
542 measured the execution time of *multiple* calls to a function in a loop, thus including potential  
543 garbage-collection effects.

544 We also measured the execution times of L1-norm minimisation and the “linear-regression”  
545  $((X^T X)^{-1} X^T y)$  similarly, but without a C implementation.

## 546 4.2.2 Hypothesis

547 We expected the C implementation to be faster than the NUMLIN one because the latter has  
548 the additional (but relatively low) overhead of dimension checks and crossing the OCaml/C  
549 FFI for each call to a CBLAS routine, even though the calls and their order are exactly the  
550 same. We expected the OCaml and Python implementations to be slower because they  
551 allocate more temporaries (so possibly less cache-friendly) and carry out more floating-point  
552 operations – the CBLAS and NUMLIN implementations use ternary kernels (coalescing steps),

```

let Many x = x in
let Many x = Many (Many x) in <exp>  ⇒  <exp>

(* fixp = fix (f, x:t, <exp> : t') *)
(*1*) let Many f = Many fixp in <body>  ⇒  let rec f x = <exp> in <body>
(*2*) let f = fixp in <body>

(*1*) let Many x = <exp> in
(*-*) let Many x = Many (Many x) in <body>  ⇒  let x = <exp> in <body>
(*2*) let Many x = Many <exp> in <body>
(*3*) (fun x : t -> <body>) <exp>

```

■ **Figure 13** Removing redundant re-bindings during translation to OCaml.

```

let rec f i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    f (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row
in
f

```

■ **Figure 14** Recursive OCaml function for a summing over an array, generated (at *compile time*) from the code in Figure 2, passed through `ocamlformat` for presentation.

553 a Cholesky decomposition (of a symmetric matrix, which is more efficient than the LU  
 554 decomposition used for inverting a matrix in OWL and NUMPY) and `symm` (symmetric matrix  
 555 multiplication, halving the number of floating-point multiplications required).

### 556 4.2.3 Results

557 The results in Figures 15 are as we expected: C is the fastest, followed by NUMLIN, with  
 558 OCaml and Python last. Differences in timings are quite pronounced at small matrix sizes,  
 559 but are still significant at larger ones. Specifically for the Kalman filter, for  $n = 625$ , CBLAS  
 560 took  $112 \pm 35\text{ ms}$ , NUMLIN took  $105 \pm 25\text{ ms}$ , OWL took  $124 \pm 38\text{ ms}$  and NUMPY took  
 561  $112 \pm 12\text{ ms}$ ; for  $n = 3125$ , CBLAS took  $10.8 \pm 0.7\text{ s}$ , NUMLIN took  $12.0 \pm 1.2\text{ s}$ , OWL took  
 562  $13.3 \pm 0.2\text{ s}$  and NUMPY took  $12.7 \pm 0.6\text{ s}$ .

563 Worth highlighting here is the other major advantage of using NUMLIN is reduced  
 564 memory usage. Whilst the OWL and NUMPY use 11 temporary matrices for the Kalman  
 565 filter, (*excluding* the 2 matrices which store the results), using  $n + n^2 + 4nk + 3k^2 + 2k \approx 4n^2$   
 566 (for  $k = 3n/5$ ) words of memory, CBLAS and NUMLIN use only 2 temporary matrices  
 567 (*excluding* the *one* matrix which stores one of the results), using only  $n^2 + nk \leq 2n^2$  words  
 568 of memory.

### 569 4.2.4 Analysis

570 As matrix sizes increase, assuming sufficient memory, the difference in the number of floating-  
 571 point operations ( $O(n^3)$ ) dominates execution times. However for small matrix sizes, since  
 572  $n$  is small and the measurements were over multiple calls to a function in a loop, the large  
 573 number of temporaries show the adverse effect of not re-using memory at even quite small  
 574 matrix sizes: creating pressure on the garbage collector.

## 575 5 Discussion and Related Work

### 576 5.1 Finding Bugs in SymPy's Output

577 Prior to this project, we had little experience with linear algebra libraries or the problem  
578 of matrix expression compilation. As such, we based our initial NUMLIN implementation  
579 of a Kalman filter using BLAS and LAPACK, on a popular GitHub gist of a Fortran  
580 implementation, one that was *automatically generated* from SymPy's matrix expression  
581 compiler [15].

582 Once we translated the implementation from Fortran to NUMLIN, we attempted to compile  
583 it and found that (to our surprise) it did not type-check. This was because the original  
584 implementation contained incorrect aliasing, unused variables and unnecessary temporaries,  
585 and did not adhere to Fortran's read/write permissions (with respect to `intent` annotations  
586 `in`, `out` and `inout`) all of which were now highlighted by NUMLIN's type system.

587 The original implementation used 6 temporaries, one of which was immediately spotted  
588 as never being used due to linearity. It also contained two variables which were marked as  
589 `intent(in)` but would have been written over by calls to 'gemv', spotted by the fractional-  
590 capabilities feature. Furthermore, it used a matrix *twice* in a call to 'symv', once with a read  
591 permission but once with a *write* permission. Fortran assumes that any parameter being  
592 written to is not aliased and so this call was not only incorrect, but illegal according to the  
593 standard, both aspects of which were captured by linearity and fractional-capabilities.

594 Lastly, it contained another unnecessary temporary, however one that was not obvious  
595 without linear types. To spot it, we first performed live-range splitting (checked by linearity)  
596 by hoisting calls to `freeM` and then annotated the freed matrices with their dimensions.  
597 After doing so and spotting two disjoint live-ranges of the same size, we replaced a call to  
598 `freeM` followed by allocating call to `copy` with one, in-place call to `copyM_to`. We believe  
599 the ability to boldly refactor code which manages memory is good evidence of the usefulness  
600 of linearity as a tool for programming.

### 601 5.2 Related Work

602 Using linear types for BLAS routines is a particularly good domain fit (given the implicit  
603 restrictions on aliasing arguments), and as a result the idea of using substructural types  
604 to express array computations is not a particularly new one [16, 10, 4]. However, many of  
605 these designs have been focused on building languages to *implement* the kernel linear algebra  
606 functions, and as a result, they tend to add additional limitations on the language design.  
607 Both Futhark [10] and Single Assignment C [16] omit higher-order functions to facilitate  
608 compilation to GPUs. The work of [4] forbids term-level recursion, in order to ensure that  
609 all higher-order computations can be statically normalized away and thereby maximize  
610 opportunities for array fusion.

611 In contrast, our approach is to begin with the assumption that we can take existing  
612 efficient BLAS-like libraries, and then enforce their correct *usage* using a linear type discipline  
613 with fractional permissions.

614 This approach is similar to the one taken in linear algebra libraries for Rust – these libraries  
615 typically take advantage of the distinction that Rust's type system offers between mutable  
616 views/references to arrays. The work of [18] and [11] suggest that Rust's borrow-checker can  
617 be expressed in simpler terms using *fractional-permissions*, though to our knowledge the  
618 programmer-visible lifetime analysis in Rust has never been formalized.

619 Working explicitly with fractional permissions has two main benefits. First, our type

system demonstrates that type systems for fractional permissions can be dramatically simpler than existing state-of-the-art approaches, including both industrial languages like Rust, as well as academic (such as those developed by [5]). Bierhoff *et al*'s type system, much like Rust's, builds a complex dataflow analysis into the typing rules to infer when variables can be shared or not. This allows for more natural-looking user programs, but can create the impression that using fractional permissions requires a heavy theoretical and engineering effort going well beyond that needed for supporting basic linear types.

Instead, our approach, of requiring sharing to be made explicit, lets us demonstrate that the existing unification machinery already in place for ordinary ML-style type inference can be reused to support fractions. Basically, we can view sharing a value as dividing a fraction by two, and after taking logarithms all fractions are Peano numbers, whose equality can be established with ordinary unification.

This fact is important because there are major upcoming implementations of linear types such as Linear Haskell [3], which do not have built-in support for fractional permissions. Instead, Linear Haskell takes a slightly different definition of linearity, one based on *arrows* as opposed to *kinds*: for  $f : a \multimap b$ , if  $f u$  is used exactly once *then*  $u$  is used exactly once. Whilst this has the advantage of being backwards-compatible, it also means that the type system has no built-in support for the concurrent reader, exclusive writer pattern that fractional permissions enable.

However, since our type system shows demonstrates unification is “all one needs” for fractions, it should be possible to *encode* NUMLIN's approach to fractional permissions in Linear Haskell by adding a GADT-style natural number index to array types tracking the fraction, which should enable supporting high-performance BLAS bindings in Linear Haskell. Actually implementing this is something we leave for future work, as there remains one issue which we do not see a good encoding for. Namely, only having support for linear functions makes it a bit inconvenient to manipulate linear values directly – programs end up taking on a CPS-like structure. This seems to remain an advantage of a direct implementation of linear types over the Linear Haskell style.

### 5.3 Simplicity and Further Work

We are pleasantly surprised at how simple the overall design and implementation of NUMLIN is, given its expressive power and usability. So simple in fact, that fractions, a convenient theoretical abstraction until this point, could be implemented by restricting division and multiplication to be by 2 only [7], thus turning any required arithmetic into unification.

Indeed, the focus on getting a working prototype early on (so that we could test it with real BLAS/LAPACK routines as soon as possible) meant that we only added features to the type system when it was clear that they were absolutely necessary: these features were !-types and value-restriction for the `Many` constructor.

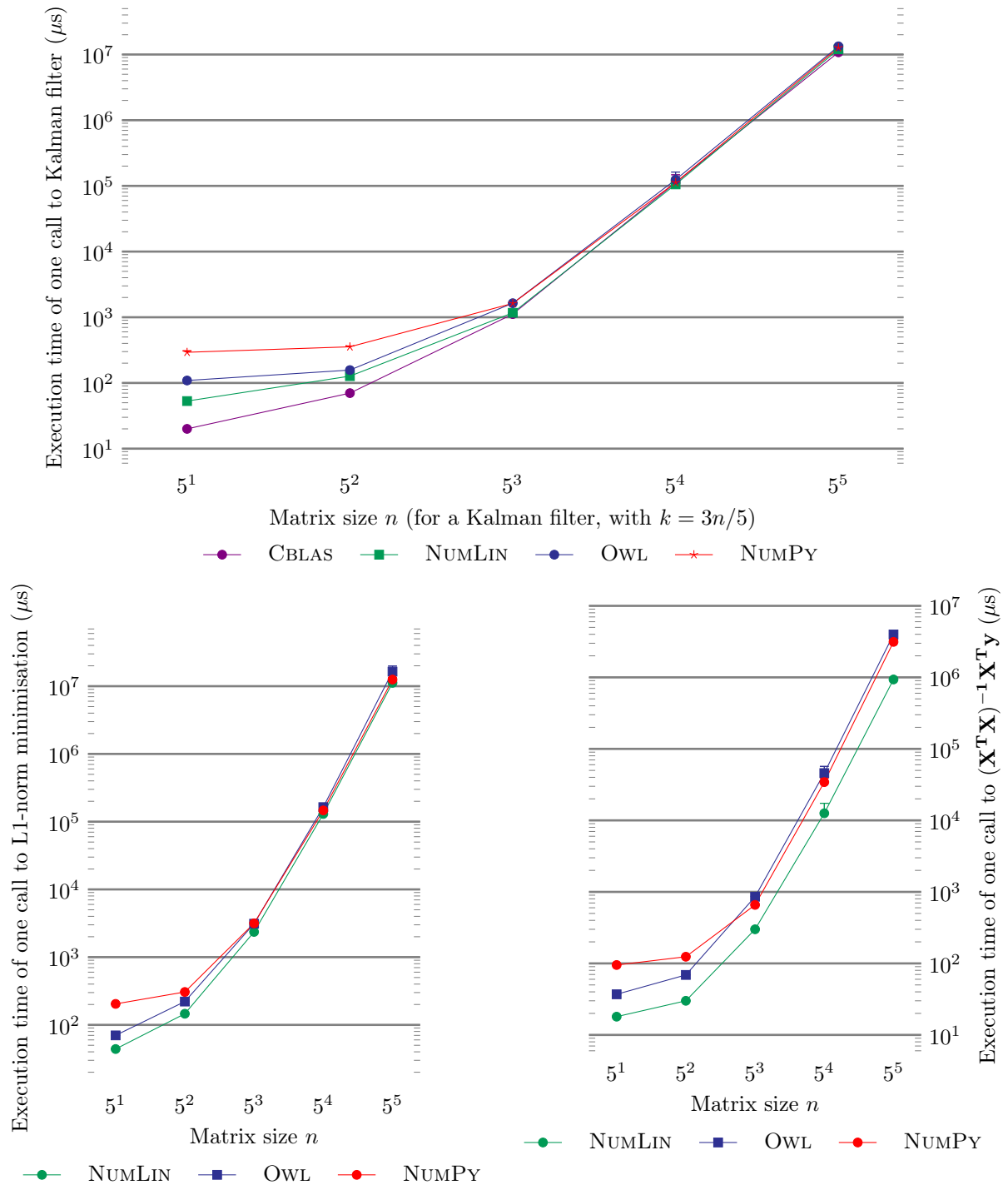
Going forwards, one may wish to eliminate even more runtime errors from NUMLIN, by extending its type system. For example, we could have used existential types to statically track pointer identities[14], or parametric polymorphism.

We could also attempt to catch mismatched dimensions at compile time as well. While this could be done with generative phantom types[1], using dependent types may offer more flexibility in *partitioning* regions[13] or statically enforcing dimensions related constraints of the arguments at compile-time. ATS[9] is an example of a language which combines linear types with a sophisticated proof layer. But although it provides BLAS bindings, it does not aim to provide aliasing restrictions as demonstrated in this paper.

Taking this idea one step even further, since matrix dimensions are typically fixed

667 at runtime, we could *stage* NUMLIN programs and compile matrix expressions using more  
668 sophisticated algorithms[2]. However, it is worth noting that without care, such algorithms[15],  
669 usually based on graph-based, ad-hoc dataflow analysis, can produce erroneous output which  
670 would not get past a linear type system with fractions.

671 We also think that this concept (and the general design of its implementation) need not  
672 be limited to linear algebra: we could conceivably ‘backport’ this idea to other contexts that  
673 need linearity (concurrency, single-use continuations, zero-copy buffer, streaming I/O) or  
674 combine it with dependent types to achieve even more expressive power to split up a single  
675 block of memory into multiple regions in an arbitrary manner[13].



**Figure 15** Comparison of execution times (error bars are present but quite small). Small matrices and timings  $n \leq 5^3$  were micro-benchmarked with the `Core_bench` library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.



---

**References**


---

- 1 Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. *arXiv preprint arXiv:1512.01898*, 2015.
- 2 Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint arXiv:1804.04021*, 2018.
- 3 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):5, 2017.
- 4 Jean-Philippe Bernardy, Victor López Juan, and Josef Svenningsson. Composable efficient array computations using linear types. *Unpublished Draft*, 2016.
- 5 Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Fraction polymorphic permission inference.
- 6 Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.
- 7 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.
- 8 Alex Bronstein, Yoni Choukroun, Ron Kimmel, and Matan Sela. Consistent discretization and minimization of the l1 norm on manifolds. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 435–440. IEEE, 2016.
- 9 Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In *International Workshop on Frontiers of Combining Systems*, pages 310–320. Springer, 2005.
- 10 Troels Henriksen, Niels GW Serup, Martin Elsmann, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *ACM SIGPLAN Notices*, 52(6):556–571, 2017.
- 11 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018. URL: <https://doi.org/10.1145/3158154>, doi:10.1145/3158154.
- 12 Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.
- 13 Conor McBride. Code mesh london 2016, keynote: Spacemonads. <https://www.youtube.com/watch?v=QojLQY5H0RI>. Accessed: 08/05/2018.
- 14 Greg Morrisett, Amal Ahmed, and Matthew Fluet. L 3: a linear language with locations. In *International Conference on Typed Lambda Calculi and Applications*, pages 293–307. Springer, 2005.
- 15 Matthew Rocklin. *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, 2013.
- 16 Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.
- 17 Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.
- 18 Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust distilled: An expressive tower of languages. *arXiv preprint arXiv:1806.02693*, 2018.

## 721 **A** NumLin Specification

### 722 **A.1** Static Semantics

723  $\boxed{\Theta; \Delta; \Gamma \vdash e : t}$  Typing rules for expressions

724 
$$\frac{}{\Theta; \Delta; \cdot, x : t \vdash x : t} \text{TY\_VAR\_LIN}$$

725 
$$\frac{x : t \in \Delta}{\Theta; \Delta; \cdot \vdash x : t} \text{TY\_VAR}$$

726 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : t \\ \Theta; \Delta; \Gamma', x : t \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let } x = e \text{ in } e' : t'} \text{TY\_LET}$$

727 
$$\frac{}{\Theta; \Delta; \cdot \vdash () : \text{unit}} \text{TY\_UNIT\_INTRO}$$

728 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : \text{unit} \\ \Theta; \Delta; \Gamma' \vdash e' : t \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let } () = e \text{ in } e' : t} \text{TY\_UNIT\_ELIM}$$

729 
$$\frac{}{\Theta; \Delta; \cdot \vdash \text{true} : \text{bool}} \text{TY\_BOOL\_TRUE}$$

730 
$$\frac{}{\Theta; \Delta; \cdot \vdash \text{false} : \text{bool}} \text{TY\_BOOL\_FALSE}$$

731 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !\text{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t} \text{TY\_BOOL\_ELIM}$$

732 
$$\frac{}{\Theta; \Delta; \cdot \vdash k : \text{int}} \text{TY\_INT\_INTRO}$$

733 
$$\frac{}{\Theta; \Delta; \cdot \vdash el : \text{elt}} \text{TY\_ELT\_INTRO}$$

734 
$$\frac{\begin{array}{l} \Theta; \Delta; \cdot \vdash v : t \\ v \neq l \end{array}}{\Theta; \Delta; \cdot \vdash \text{Many } v : !t} \text{TY\_BANG\_INTRO}$$

735 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \text{let Many } x = e \text{ in } e' : t'} \text{TY\_BANG\_ELIM}$$

736 
$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e : t \\ \Theta; \Delta; \Gamma' \vdash e' : t' \end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash (e, e') : t \otimes t'} \text{TY\_PAIR\_INTRO}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_{12} : t_1 \otimes t_2 \quad \Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e : t}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let} (a, b) = e_{12} \mathbf{in} e : t} \quad \text{TY\_PAIR\_ELIM}$$

$$\frac{\Theta \vdash t' \text{ Type} \quad \Theta; \Delta; \Gamma, x : t' \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun} x : t' \rightarrow e : t' \multimap t} \quad \text{TY\_LAMBDA}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t' \multimap t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash e e' : t} \quad \text{TY\_APP}$$

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun} fc \rightarrow e : \forall fc. t} \quad \text{TY\_GEN}$$

$$\frac{\Theta \vdash f \text{ Perm} \quad \Theta; \Delta; \Gamma \vdash e : \forall fc. t}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \quad \text{TY\_SPC}$$

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix} (g, x : t, e : t') : t \multimap t'} \quad \text{TY\_FIX}$$

743

## 744 A.2 Dynamic Semantics

$$\boxed{\langle \sigma, e \rangle \rightarrow \text{Config}} \quad \text{Operational semantics}$$

$$\frac{}{\langle \sigma, \mathbf{let} () = () \mathbf{in} e \rangle \rightarrow \langle \sigma, e \rangle} \quad \text{OP\_LET\_UNIT}$$

$$\frac{}{\langle \sigma, \mathbf{let} x = v \mathbf{in} e \rangle \rightarrow \langle \sigma, e[x/v] \rangle} \quad \text{OP\_LET\_VAR}$$

$$\frac{}{\langle \sigma, \mathbf{if} (\mathbf{Many} \text{ true}) \text{ then } e_1 \text{ else } e_2 \rangle \rightarrow \langle \sigma, e_1 \rangle} \quad \text{OP\_IF\_TRUE}$$

$$\frac{}{\langle \sigma, \mathbf{if} (\mathbf{Many} \text{ false}) \text{ then } e_1 \text{ else } e_2 \rangle \rightarrow \langle \sigma, e_2 \rangle} \quad \text{OP\_IF\_FALSE}$$

$$\frac{}{\langle \sigma, \mathbf{let} \mathbf{Many} x = \mathbf{Many} v \mathbf{in} e \rangle \rightarrow \langle \sigma, e[x/v] \rangle} \quad \text{OP\_LET\_MANY}$$

$$\frac{}{\langle \sigma, \mathbf{let} (a, b) = (v_1, v_2) \mathbf{in} e \rangle \rightarrow \langle \sigma, e[a/v_1][b/v_2] \rangle} \quad \text{OP\_LET\_PAIR}$$

$$\frac{}{\langle \sigma, (\mathbf{fun} fc \rightarrow v)[f] \rangle \rightarrow \langle \sigma, e[fc/f], v[fc/f] \rangle} \quad \text{OP\_FRAC\_PERM}$$

$$\frac{}{\langle \sigma, \mathbf{fix} (g, x : t, e : t') v \rangle \rightarrow \langle \sigma, e[x/v][g/\mathbf{fix} (g, x : t, e : t')] \rangle} \quad \text{OP\_APP\_FIX}$$

$$\frac{}{\langle \sigma, (\mathbf{fun} x : t \rightarrow e) v \rangle \rightarrow \langle \sigma, e[x/v] \rangle} \quad \text{OP\_APP\_LAMBDA}$$

754

## 23:26 NumLin: Linear Types for Linear Algebra

$$\begin{array}{l}
\text{755} \quad \frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow \langle \sigma', C[e'] \rangle} \quad \text{OP\_CONTEXT} \\
\text{756} \quad \frac{\langle \sigma, e \rangle \rightarrow \mathbf{err}}{\langle \sigma, C[e] \rangle \rightarrow \mathbf{err}} \quad \text{OP\_CONTEXT\_ERR} \\
\text{757} \quad \frac{0 \leq k_1, k_2 \quad l \text{ fresh}}{\langle \sigma, \mathbf{matrix} \, k_1 \, k_2 \rangle \rightarrow \langle \sigma + \{l \mapsto_1 M_{k_1, k_2}\}, l \rangle} \quad \text{OP\_MATRIX} \\
\text{758} \quad \frac{k_1 < 0 \text{ or } k_2 < 0}{\langle \sigma, \mathbf{matrix} \, k_1 \, k_2 \rangle \rightarrow \mathbf{err}} \quad \text{OP\_MATRIX\_NEG} \\
\text{759} \quad \frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1, k_2}\}, \mathbf{free} \, l \rangle \rightarrow \langle \sigma, () \rangle} \quad \text{OP\_FREE} \\
\text{760} \quad \frac{}{\langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, \mathbf{share}[f] \, l \rangle \rightarrow \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, (l, l) \rangle} \quad \text{OP\_SHARE} \\
\text{761} \quad \frac{}{\langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\}, \mathbf{unshare}[f] \, l \, l \rangle \rightarrow \langle \sigma + \{l \mapsto_f m_{k_1, k_2}\}, l \rangle} \quad \text{OP\_UNSHARE\_EQ} \\
\text{762} \quad \frac{l \neq l'}{\langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1, k_2}\} + \{l' \mapsto_{\frac{1}{2}f} m'_{k_1, k_2}\}, \mathbf{unshare}[f] \, l \, l' \rangle \rightarrow \mathbf{err}} \quad \text{OP\_UNSHARE\_NEQ} \\
\text{763} \quad \frac{\begin{array}{l} \sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1 \, k_1, k_2}\} + \{l_2 \mapsto_{fc_2} m_{2 \, k_2, k_3}\} \\ \sigma_1 \equiv \sigma' + \{l_3 \mapsto_1 m_{3 \, k_1, k_3}\} \\ \sigma_2 \equiv \sigma' + \{l_3 \mapsto_1 (m_1 \, m_2 + m_3)_{k_1, k_3}\} \end{array}}{\langle \sigma_1, \mathbf{gemm}[fc_1] \, l_1[fc_2] \, l_2 \, l_3 \rangle \rightarrow \langle \sigma_2, ((l_1, l_2), l_3) \rangle} \quad \text{OP\_GEMM\_MATCH} \\
\text{764} \quad \frac{k_2 \neq k'_2 \quad \sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1 \, k_1, k_2}\} + \{l_2 \mapsto_{fc_2} m_{2 \, k'_2, k_3}\}}{\langle \sigma' + \{l_3 \mapsto_1 m_{1 \, k_1, k_3}\}, \mathbf{gemm}[fc_1] \, l_1[fc_2] \, l_2 \, l_3 \rangle \rightarrow \mathbf{err}} \quad \text{OP\_GEMM\_MISMATCH} \\
\text{765}
\end{array}$$

766

**B** Interpretation

767

**B.1** Definitions

768

Operationally,  $\text{Heap} \sqsubseteq \text{Loc} \times \text{Permission} \times \text{Matrix}$  (a multiset), denoted with a  $\sigma$ .

769

Define its *interpretation* to be  $\text{Loc} \rightarrow \text{Permission} \times \text{Matrix}$  with  $\star : \text{Heap} \times \text{Heap} \rightarrow \text{Heap}$  as

770

follows:

$$771 \quad (\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \wedge l \notin \text{dom}(\varsigma_2) \\ \varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \wedge l \notin \text{dom}(\varsigma_1) \\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \wedge (f_2, m) = \varsigma_2(l) \wedge f_1 + f_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

772

Commutativity and associativity of  $\star$  follows from that of  $+$ .

773

$\varsigma_1 \star \varsigma_2$  is *defined* if it is for all  $l \in \text{dom}(\varsigma_1) \cup \text{dom}(\varsigma_2)$ .

774

Define  $\mathcal{H}[\sigma] = \star_{(l,f,m) \in \sigma} [l \mapsto_f m]$  and **implicitly denote**  $\varsigma \equiv \mathcal{H}[\theta(\sigma)]$ .

775

776

The  $n$ -fold iteration for the  $\rightarrow$  (functional) relation, is also a (functional) relation:

777  
778

$$\forall n. \mathbf{err} \rightarrow^n \mathbf{err} \quad \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle \quad \langle \sigma, e \rangle \rightarrow^0 \langle \sigma, e \rangle \quad \langle \sigma, e \rangle \rightarrow^{n+1} ((\langle \sigma, e \rangle \rightarrow) \rightarrow^n)$$

779

Hence, all bounded iterations end in either an **err**, a heap-and-expression or a heap-and-value.

## 780 B.2 Interpretation

$$781 \quad \mathcal{V}_k[\mathbf{unit}] = \{(\emptyset, *)\}$$

782

$$783 \quad \mathcal{V}_k[\mathbf{bool}] = \{(\emptyset, true), (\emptyset, false)\}$$

784

$$785 \quad \mathcal{V}_k[\mathbf{int}] = \{(\emptyset, n) \mid 2^{-63} \leq n \leq 2^{63} - 1\}$$

786

$$787 \quad \mathcal{V}_k[\mathbf{elt}] = \{(\emptyset, f) \mid f \text{ a IEEE Float64}\}$$

788

$$789 \quad \mathcal{V}_k[f \mathbf{mat}] = \{(\{l \mapsto_{2^{-f}} \_ \}, l)\}$$

790

$$791 \quad \mathcal{V}_k[!t] = \{(\emptyset, \mathbf{Many} \ v) \mid (\emptyset, v) \in \mathcal{V}_k[t]\}$$

792

$$793 \quad \mathcal{V}_k[\forall fc. t] = \{(\varsigma, \mathbf{fun} \ fc \rightarrow v) \mid \forall f. (\varsigma[fc/f], v[fc/f]) \in \mathcal{V}_{k-1}[t[fc/f]]\}$$

794

$$795 \quad \mathcal{V}_k[t_1 \otimes t_2] = \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[t_1] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[t_2]\}$$

796

$$797 \quad \mathcal{V}_k[t' \multimap t] = \{(\varsigma_v, v) \mid (v \equiv \mathbf{fun} \ x : t' \rightarrow e \vee v \equiv \mathbf{fix}(g, x : t', e : t)) \wedge$$

$$798 \quad \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[t']. \varsigma_v \star \varsigma'_v \text{ defined} \Rightarrow (\varsigma_v \star \varsigma'_v, v v') \in \mathcal{C}_j[t]\}$$

799

$$800 \quad \mathcal{C}_k[t] = \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r. \varsigma_s \star \varsigma_r \text{ defined} \Rightarrow \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f.$$

$$801 \quad \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}[t])\}$$

802

$$803 \quad \mathcal{I}_k[\cdot]\theta = \{\emptyset\}$$

804

$$805 \quad \mathcal{I}_k[\Delta, x : t]\theta = \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\Delta]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\theta(t)]\}$$

806

$$807 \quad \mathcal{L}_k[\cdot]\theta = \{(\emptyset, \emptyset)\}$$

808

$$809 \quad \mathcal{L}_k[\Gamma, x : t]\theta = \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\theta(t)]\}$$

810

$$811 \quad \mathcal{H}[\sigma] = \star_{(l, f, m) \in \sigma} [l \mapsto_f m]$$

$$812 \quad \varsigma \equiv \mathcal{H}[\theta(\sigma)]$$

813

$$814 \quad {}_k[\Theta; \Delta; \Gamma \vdash e : t] = \forall \theta, \delta, \gamma, \sigma. \Theta = \text{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma]\theta \wedge \delta \in \mathcal{I}_k[\Delta]\theta \Rightarrow$$

$$816 \quad (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)]$$

## 817 **C** Lemmas

818 **C.1**  $\forall \sigma_s, \sigma_r, e. \varsigma_s \star \varsigma_r \text{ defined} \Rightarrow \forall n. \langle \sigma_s, e \rangle \rightarrow^n = \langle \sigma_s + \sigma_r, e \rangle \rightarrow^n$

819 SUFFICES: By induction on  $n$ , consider only the cases  $\langle \sigma_s, e \rangle \rightarrow \langle \sigma_f, e_f \rangle$  where  $\sigma_s \neq \sigma_f$ .

820  
821 PROOF SKETCH: Only  $\text{OP\_}\{\text{FREE, MATRIX, SHARE, UNSHARE\_EQ, GEMM\_MATCH}\}$   
822 change the heap: the rest are either parametric in the heap or step to an **err**.

823  
824 PROVE:  $\langle \sigma_s + \sigma_r, e \rangle \rightarrow \langle \sigma_f + \sigma_r, e_f \rangle$ .

825  
826  $\langle 1 \rangle 1$ . CASE:  $\text{OP\_FREE}$ ,  $\sigma_s \equiv \sigma' + \{l \mapsto_1 m\}$ ,  $\sigma_f = \sigma'$ .

827 PROOF: Instantiate  $\text{OP\_FREE}$  with  $(\sigma' + \sigma_r) + \{l \mapsto_1 m\}$ ,  
828 valid because  $l \notin \text{dom}(\varsigma_r)$  by  $\varsigma' \star [l \mapsto_1 m] \star \varsigma_r$  defined (assumption).

829  $\langle 1 \rangle 2$ . CASE:  $\text{OP\_MATRIX}$

830 PROOF: Rule has no requirements on  $\sigma_s$  so will also work with  $\sigma_s + \sigma_r$ .

831  $\langle 1 \rangle 3$ . CASE:  $\text{OP\_SHARE}$ ,  $\sigma_s \equiv \sigma' + \{l \mapsto_f m\}$ ,  $\sigma_f = \sigma' + \{l \mapsto_{\frac{1}{2}.f} m\} + \{l \mapsto_{\frac{1}{2}.f} m\}$ .

832 PROOF: Union-ing  $\sigma_r$  does not remove  $l \mapsto_f m$ , so that can be split out of  $\sigma_s + \sigma_r$  as  
833 before.

834  $\langle 1 \rangle 4$ . CASE:  $\text{OP\_UNSHARE\_EQ}$ ,  $\sigma_s \equiv \sigma' + \{l \mapsto_{\frac{1}{2}.f} m\} + \{l \mapsto_{\frac{1}{2}.f} m\}$ ,  $\sigma_f = \sigma' + \{l \mapsto_f m\}$ .

835  $\langle 2 \rangle 1$ . Union-ing  $\sigma_r$  does not remove  $l \mapsto_{\frac{1}{2}.f} m$ , so that can still be split out of  $\sigma_s + \sigma_r$ .

836  $\langle 2 \rangle 2$ . There may also be other valid splits introduced by  $\sigma_r$ .

837  $\langle 2 \rangle 3$ . However, by assumption of  $\varsigma_s \star \varsigma_r$  defined, any splitting of  $\sigma_s + \sigma_r$  will satisfy  
838  $f \leq 1$ .

839  $\langle 1 \rangle 5$ . CASE:  $\text{OP\_GEMM\_MATCH}$

840  $\langle 2 \rangle 1$ . By assumption of  $\varsigma_s \star \varsigma_r$  defined, either  $l_1$  (or  $l_2$ , or both) are not in  $\sigma_r$ , or they  
841 are and the matrix values they point to are the same.

842  $\langle 2 \rangle 2$ . The permissions (of  $l_1$  and/or  $l_2$ ) may differ, but  $\text{OP\_GEMM\_MATCH}$  universally  
843 quantifies over them and leaves them unchanged, so they are irrelevant.

844  $\langle 2 \rangle 3$ . Only the pointed to matrix value at  $l_3$  changes.

845  $\langle 2 \rangle 4$ . SUFFICES:  $l_3 \notin \pi_1[\sigma_r]$ .

846  $\langle 2 \rangle 5$ . By assumption of  $\varsigma_s \star \varsigma_r$  defined,  $l_3 \notin \text{dom}(\varsigma_r)$ .

847  $\langle 2 \rangle 6$ . Hence  $l_3 \notin \pi_1[\sigma_r]$ .

848 **C.2**  $\forall k, t. \mathcal{V}_k[t] \subseteq \mathcal{C}_k[t]$

849 Follows from definition of  $\mathcal{C}_k[t]$ ,  $\rightarrow^j$  ( $\forall n. \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle$ ) for arbitrary  $j \leq k$  and C.1.



850 **C.3**  $\forall \theta, \delta, \gamma, v. \theta(\delta(\gamma(v)))$  is a value.

851  $\theta$  is irrelevant because it only maps fractional permission variables to fractional permissions.

852 By construction,  $\delta$  and  $\gamma$  only map variables to values, and values are closed under substitution.

853 **C.4**  $\forall k, \sigma, \sigma', e, e', t. (\varsigma', e') \in \mathcal{C}_k[t] \wedge \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \Rightarrow (\varsigma, e) \in \mathcal{C}_{k+1}[t]$

854 In the lemma, and for the rest of its proof,  $\varsigma = \mathcal{H}[\sigma]$ .

855 ASSUME: arbitrary  $j < k + 1$ , and  $\sigma_r$  such that  $\varsigma \star \varsigma_r$  defined.

856

857 (1)1. CASE:  $j = 0$ . Clearly  $\sigma_f = \sigma_s + \sigma_r$  and  $e' = e$ .

858 Remains to show that if  $e$  is a value then  $(\varsigma_s \star \varsigma_r, e) \in \mathcal{V}_k[t]$ .

859 This is true vacuously, because by assumption,  $e$  is not a value.

860 (1)2. CASE:  $j \geq 1$ . We have  $\langle \sigma, e \rangle \rightarrow^j = \langle \sigma', e' \rangle \rightarrow^{j-1}$ .

861 Instantiate  $(\varsigma', e') \in \mathcal{C}_k[t]$ , with  $j - 1 < k$  and  $\sigma_r$  to conclude the required conditions.

862 **C.5**  $j \leq k \Rightarrow \_k[\cdot] \subseteq \_j[\cdot]$

863 For the rest of this proof,  $\varsigma = \mathcal{H}[\sigma]$ .

864 Lemma C.4 is the inductive step for this lemma for the  $\mathcal{C}[\cdot]$  case.

865 Need to prove for  $\mathcal{V}[\cdot]$ , by induction on  $t$  and then index.

866 SUFFICES: Consider only  $t \multimap t'$  case, rest use  $k$  directly on structure of type.

867 ASSUME: Arbitrary  $j \leq k$  and  $(\varsigma_{v'}, v') \in \mathcal{V}_k[t \multimap t']$ .

868 PROVE:  $(\varsigma_{v'}, v') \in \mathcal{V}_j[t \multimap t']$ .

869

870 (1)1.  $v'$  is of the correct syntactic form (lambda or fixpoint) by assumption.

871 (1)2. ASSUME: arbitrary  $j' \leq j$  and  $(\varsigma_v, v) \in \mathcal{V}_{j'}[t]$  such that  $\varsigma_{v'} \star \varsigma_v$  is defined.

872 (1)3. SUFFICES: to show  $(\varsigma_{v'} \star \varsigma_v, v'v) \in \mathcal{C}_{j'}[t']$ .

873 (1)4. This is true by instantiating  $(\varsigma_{v'}, v') \in \mathcal{V}_k[t \multimap t']$  with  $j' \leq k$  and  $(\varsigma_v, v) \in \mathcal{V}_{j'}[t]$ .

874 **C.6**  $\forall \Delta, \Gamma, t, k, \theta, \delta, \gamma. \delta \in \mathcal{I}_k[\Delta]\theta \wedge \gamma \in \pi_2[\mathcal{L}_k[\Gamma]\theta] \Rightarrow \text{dom}(\Delta) = \text{dom}(\delta)$   
 875 **and**  $\text{dom}(\Gamma) = \text{dom}(\gamma)$

876 PROOF: By induction on  $\Delta$  and  $\Gamma$ .

877 **C.7**  $\forall k, \Gamma, \Gamma', \theta, \sigma_+, \gamma_+. (\varsigma_+, \gamma) \in \mathcal{L}_k[\Gamma, \Gamma']\theta \wedge \Gamma, \Gamma' \text{ disjoint} \Rightarrow$

878  $\exists \sigma, \gamma, \sigma', \gamma'. \sigma_+ = \sigma + \sigma' \wedge \gamma, \gamma' \text{ disjoint} \wedge \gamma_+ = \gamma \cup \gamma' \wedge (\varsigma, \gamma) \in$

879  $\mathcal{L}_k[\Gamma] \wedge (\varsigma', \gamma') \in \mathcal{L}_k[\Gamma']$

880 PROOF: By induction on  $\Gamma'$ .

881 **C.8**  $\forall e, \sigma, e', \sigma', \theta. \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \Rightarrow \langle \theta(\sigma), \theta(e) \rangle \rightarrow \langle \theta(\sigma'), \theta(e') \rangle$

882 PROOF: By induction on  $\rightarrow$ .

883  $\langle 1 \rangle 1$ . ASSUME: Arbitrary  $e, \sigma, e', \sigma', \theta$  such that  $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$ .

884  $\langle 1 \rangle 2$ . SUFFICES: To consider only the following rules which mention fractional permission  
885 variables.

886 OP\_FRAC\_PERM, OP\_SHARE, OP\_UNSHARE\_(N)EQ and OP\_GEMM\_(MIS)MATCH.

887  $\langle 1 \rangle 3$ . CASE: OP\_FRAC\_PERM.

888 Because substitution avoids capture,

889  $\langle \theta(\sigma), \theta((\mathbf{fun} \, fc \rightarrow v) [f]) \rangle \rightarrow \langle \theta(\sigma' [fc/f]), \theta(v [fc/f]) \rangle$ .

890  $\langle 1 \rangle 4$ . The rest of the cases are parametric in their use of fractional permission variables and  
891 so will take the same step after any substitution.

892  $\langle 1 \rangle 5$ . COROLLARY: If  $\langle \sigma [fc/f_1], e [fc/f_1] \rangle \rightarrow^n \langle \sigma_2, e'_2 \rangle$  and  $\langle \sigma [fc/f_2], e [fc/f_2] \rangle \rightarrow^n \langle \sigma_2, e'_2 \rangle$ ,  
893 then  $\exists \sigma, e'. \sigma_1 = \sigma [fc/f_1] \wedge \sigma_2 = \sigma [fc/f_2] \wedge e'_1 = e' [fc/f_1] \wedge e'_2 = e' [fc/f_2]$ .

## 894 **D** Soundness

895  $\forall \Theta, \Delta, \Gamma, e, t. \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k. {}_k \llbracket \Theta; \Delta; \Gamma \vdash e : t \rrbracket$

896 PROOF SKETCH: Induction over the typing judgements.

897

898 ASSUME: 1. Arbitrary  $\Theta, \Delta, \Gamma, e, t$  such that  $\Theta; \Delta; \Gamma \vdash e : t$ .

899 2. Arbitrary  $k, \theta, \delta, \gamma, \sigma$  such that:

900 a.  $\Theta = \text{dom}(\theta)$

901 b.  $\delta \in \mathcal{I}_k \llbracket \Delta \rrbracket \theta$ .

902 c.  $(\varsigma, \gamma) \in \mathcal{L}_k \llbracket \Gamma \rrbracket \theta$

903 3. W.l.o.g., all variables are distinct, hence  $\Theta$ ,  $\text{dom}(\Delta)$  and  $\text{dom}(\Gamma)$  are disjoint so  
904 order of  $\theta$ ,  $\delta$  and  $\gamma$  (as substitutions defined recursively over expressions) is  
905 irrelevant.

906

907 PROVE:  $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k \llbracket \theta(t) \rrbracket$ .

908 ASSUME: Arbitrary  $j < k$  and  $\sigma_r$ , such that  $\varsigma \star \sigma_r$  defined.

909 SUFFICES:  $\langle \sigma + \sigma_r, e \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \langle \sigma + \sigma_r, e \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle$

910  $\wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f \star \sigma_r, e_f) \in \mathcal{V}_{k-j} \llbracket t \rrbracket)$ .

911 SUFFICES: By C.1, to show  $\langle \sigma, e \rangle \rightarrow^j \mathbf{err} \vee \exists \sigma_f, e_f. \langle \sigma, e \rangle \rightarrow^j \langle \sigma_f, e_f \rangle$

912  $\wedge (e_f \text{ is a value} \Rightarrow (\varsigma_f, e_f) \in \mathcal{V}_{k-j} \llbracket t \rrbracket)$

913

914  $\langle 1 \rangle 1$ . CASE: TY\_LET.

915  $\langle 2 \rangle 1$ . By induction,

916 1.  $\forall k. {}_k \llbracket \Theta; \Delta; \Gamma \vdash e : t \rrbracket$

917 2.  $\forall k. {}_k \llbracket \Theta; \Delta; \Gamma', x : t \vdash e' : t' \rrbracket$ .

- 918       $\langle 2 \rangle 2$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):  
 919              1.  $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\Gamma]$   
 920              2.  $\gamma = \gamma_e \cup \gamma_{e'}$   
 921              3.  $\sigma = \sigma_e + \sigma_{e'}$ .
- 922       $\langle 2 \rangle 3$ . So, using  $k, \theta, \delta, \gamma_e, \sigma_e$ , we have  $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\theta(t)]$ .
- 923       $\langle 2 \rangle 4$ . By  $\langle 2 \rangle 2$  ( $\gamma = \gamma_e \cup \gamma_{e'}$ ), have  $(\varsigma_e, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)]$ .
- 924       $\langle 2 \rangle 5$ . By definition of  $\mathcal{C}_k[\cdot]$  and  $\langle 2 \rangle 2$ , we instantiate with  $j$  and  $\sigma_r = \sigma_{e'}$  to conclude  
 925      that  
 926       $\langle \theta(\sigma), \theta(\delta(\gamma(e))) \rangle$  either takes  $j$  steps to **err** or another heap-and-expression  
 927       $\langle \sigma_f, e_f \rangle$ .
- 928       $\langle 2 \rangle 6$ . CASE:  $j$  steps to **err**  
 929      By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j < k$  steps.
- 930       $\langle 2 \rangle 7$ . CASE:  $j$  steps to another heap-and-expression.  
 931      If it is not a value, then OP\_CONTEXT runs  $j$  times and we are done.
- 932       $\langle 2 \rangle 8$ . If it is, then  $\exists i \leq j$ .  $(\varsigma_f, v_1) \in \mathcal{V}_{k-i}[\theta(t_1)] \subseteq \mathcal{V}_{k-j}[\theta(t_1)]$  by C.3 and C.5.  
 933      So, OP\_CONTEXT runs  $i$  times, and then we have the following.  
 934      SUFFICES:  $(\varsigma_f \star \varsigma_{e'}, \mathbf{let} \ x = v \ \mathbf{in} \ \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\theta(t')]$  by C.4  $i$  times.  
 935      SUFFICES:  $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[x/v]) \in \mathcal{C}_{k-i-1}[\theta(t')]$  by C.4.
- 936       $\langle 2 \rangle 9$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}[x \mapsto v]) \in \mathcal{L}_k[\Gamma', x : t]\theta \subseteq \mathcal{L}_{k-i-1}[\Gamma', x : t]\theta$ .
- 937       $\langle 2 \rangle 10$ . Instantiate 2 of step  $\langle 2 \rangle 1$  with  $k - i - 1, \theta, \delta, \gamma_{e'}[x \mapsto v], \sigma_{e'}$  to conclude  
 938       $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}[x \mapsto v](e')))) \in \mathcal{C}_{k-i-1}[\theta(t')]$ .
- 939       $\langle 2 \rangle 11$ . By 3, we have  $\theta(\delta(\gamma(e')))[x/v] = \theta(\delta(\gamma_{e'}[x \mapsto v](e')))$  and  
 940      by C.1 we conclude  $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[x/v]) \in \mathcal{C}_{k-i-1}[\theta(t')]$ .
- 941       $\langle 1 \rangle 2$ . CASE: TY\_PAIR\_ELIM.  
 942      PROOF SKETCH: Similar to TY\_LET, but with the following key differences.
- 943       $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\theta(t_1) \otimes \theta(t_2)]$ , we have  $v = (v_1, v_2)$ .
- 944       $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\theta(t')]$  by C.4  $i + 1$  times.
- 945       $\langle 2 \rangle 3$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2]) \in \mathcal{L}_k[\Gamma', a : t_1, b : t_2]\theta \subseteq \mathcal{L}_{k-i-1}[\Gamma', a : t_1, b : t_2]\theta$ .  
 946       $t_2]\theta$ .
- 947       $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e' : t']$  with  $\theta, \delta, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2], \sigma_{e'}$ .
- 948       $\langle 2 \rangle 5$ . By 3 (for  $\gamma = \gamma_e \cup \gamma_{e'}$  and  $a, b$ ), conclude  $(\varsigma_{e'}, \theta(\delta(\gamma(e'[a/v_1][b/v_2]))) \in \mathcal{C}_{k-i-1}[\theta(t')]$ .
- 949       $\langle 1 \rangle 3$ . CASE: TY\_BANG\_ELIM.  
 950      PROOF SKETCH: Similar to TY\_LET, but with the following key differences.
- 951       $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\theta(!t)]$ , since  $\mathcal{V}_{k-i}[\theta(!t)] = \mathcal{V}_{k-i}[\theta(!t)]$ ,  
 952      we have  $\varsigma_f = \emptyset$  and  $v = \mathbf{Many} \ v'$  for some  $(\emptyset, v') \in \mathcal{V}_{k-i}[\theta(t)]$ .

- 953  $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \text{let Many } x = \text{Many } v' \text{ in } \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\![\theta(t)]\!]$ .
- 954  $\langle 2 \rangle 3$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[x/v]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$  by C.4  $i + 1$  times.
- 955  $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\![\Theta; \Delta, x : t, \Gamma' \vdash e' : t']\!]$  with  $\theta, \delta_{e'} = \delta[x \mapsto v'], \gamma_{e'}, \sigma_{e'}$ .
- 956  $\langle 2 \rangle 5$ . By 3,  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[x/v]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$ .
- 957  $\langle 1 \rangle 4$ . CASE: TY\_UNIT\_ELIM.
- 958 PROOF SKETCH: Similar to TY\_LET, but with the following key differences.
- 959  $\langle 2 \rangle 1$ . When  $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\![\text{unit}]\!]$ , we have  $\varsigma_f = \emptyset$  and  $v = ()$ .
- 960  $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$  by C.4  $i + 1$  times.
- 961  $\langle 2 \rangle 3$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\![\Gamma']\!]\theta \subseteq \mathcal{L}_{k-i-1}[\![\Gamma']\!]\theta$ .
- 962  $\langle 2 \rangle 4$ . Instantiate  $_{k-i-1}[\![\Theta; \Delta; \Gamma' \vdash e' : t']\!]$  with  $\theta, \delta, \gamma_{e'}, \sigma_{e'}$ .
- 963  $\langle 2 \rangle 5$ . By 3  $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$ .
- 964  $\langle 1 \rangle 5$ . CASE: TY\_BOOL\_ELIM.
- 965 PROOF SKETCH: Similar to TY\_UNIT\_ELIM but with  $\text{OP\_IF\_}\{\text{TRUE}, \text{FALSE}\}$ ,  $\varsigma_f = \emptyset$
- 966 and  $v = \text{Many true}$  or  $v = \text{Many false}$ .
- 967  $\langle 1 \rangle 6$ . CASE: TY\_BANG\_INTRO.
- 968  $\langle 2 \rangle 1$ . We have,  $e = v$  for some value  $v \neq l$ ,  $\Gamma = \emptyset$  and so
- 969  $\forall k. \_k[\![\Theta; \Delta; \cdot \vdash v : t]\!]$  by induction.
- 970  $\langle 2 \rangle 2$ . SUFFICES:  $(\emptyset, \text{Many } \theta(\delta(v))) \in \mathcal{C}_k[\![! \theta(t)]\!]$  by 2c ( $\varsigma = \emptyset, \gamma = []$ ).
- 971  $\langle 2 \rangle 3$ . Instantiate  $_{k-1}[\![\Theta; \Delta; \cdot \vdash v : t]\!]$  with  $\theta, \delta, \gamma = [], \sigma = \emptyset$  to obtain  $(\emptyset, \theta(\delta(v))) \in$
- 972  $\mathcal{C}_k[\![\theta(t)]\!]$ .
- 973  $\langle 2 \rangle 4$ . Instantiate  $(\emptyset, \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$  with  $j = 0, \sigma_r = \emptyset$  and C.3 ( $\theta(\delta(v))$  is a value),
- 974 to conclude  $(\emptyset, \theta(\delta(v))) \in \mathcal{V}_k[\![\theta(t)]\!]$ .
- 975  $\langle 2 \rangle 5$ . By definition of  $\mathcal{V}_k[\![! \theta(t)]\!]$ , C.3 and C.2 we have  $(\emptyset, \text{Many } \theta(\delta(v))) \in \mathcal{C}_k[\![! \theta(t)]\!]$ .
- 976  $\langle 1 \rangle 7$ . CASE: TY\_PAIR\_INTRO.
- 977  $\langle 2 \rangle 1$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):
- 978 1.  $(\varsigma_1, \gamma_1) \in \mathcal{L}_k[\![\Gamma_1]\!]$
- 979 2.  $(\varsigma_2, \gamma_2) \in \mathcal{L}_k[\![\Gamma_2]\!]$
- 980 3.  $\gamma = \gamma_1 \cup \gamma_2$
- 981 4.  $\sigma = \sigma_1 + \sigma_2$ .
- 982  $\langle 2 \rangle 2$ . By induction,
- 983 1.  $\forall k. \_k[\![\Theta; \Delta; \Gamma_1 \vdash e_1 : t_1]\!]$
- 984 2.  $\forall k. \_k[\![\Theta; \Delta; \Gamma_2 \vdash e_2 : t_2]\!]$ .
- 985  $\langle 2 \rangle 3$ . Instantiate the first with  $k, \theta, \delta, \gamma_1, \sigma_1$ .

986        $\langle 2 \rangle 4$ . By that and  $\langle 2 \rangle 1$ ,  $(\varsigma_1, \theta(\delta(\gamma_1(e_1)))) = (\varsigma_1, \theta(\delta(\gamma(e_1)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .  
 987        $\langle 2 \rangle 5$ . So,  $\langle \theta(\sigma_1 + \sigma_2), \theta(\delta(\gamma_1(e_1))) \rangle$  either takes  $j$  steps to **err** or a heap-and-expression  
 988        $\langle \sigma_{1f}, e_{1f} \rangle$ .  
 989        $\langle 2 \rangle 6$ . CASE:  $j$  steps to **err**  
 990       By `OP_CONTEXT_ERR`, the whole expression reduces to **err** in  $j < k$  steps.  
 991        $\langle 2 \rangle 7$ . CASE:  $j$  steps to another heap-and-expression.  
 992       If it is not a value, then `OP_CONTEXT` runs  $j$  times and we are done.  
 993        $\langle 2 \rangle 8$ . If it is, then  $\exists i_1 \leq j$ .  $(\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!]$  by C.3 and C.5.  
 994       So, `OP_CONTEXT` runs  $i_1$  times, and then we have the following.  
 995       SUFFICES: By C.4,  $(\varsigma_{1f} \star \varsigma_2, (v_1, e_2)) \in \mathcal{C}_{k-i_1}[\![\theta(t_1 \otimes t_2)]\!]$ .  
 996        $\langle 2 \rangle 9$ . Instantiate the second IH with  $k, \theta, \delta, \gamma_2, \sigma_2$ .  
 997        $\langle 2 \rangle 10$ . So,  $\langle \theta(\sigma_{1f} + \sigma_2), \theta(\delta(\gamma_2(e_2))) \rangle$  either takes  $j$  steps to **err** or a heap-and-expression  
 998        $\langle \sigma_{2f}, e_{2f} \rangle$ .  
 999        $\langle 2 \rangle 11$ . CASE:  $j$  steps to **err**  
 1000       By `OP_CONTEXT_ERR`, the whole expression reduces to **err** in  $j < k$  steps.  
 1001        $\langle 2 \rangle 12$ . CASE:  $j$  steps to another heap-and-expression.  
 1002       If it is not a value, then `OP_CONTEXT` runs  $j$  times and we are done.  
 1003        $\langle 2 \rangle 13$ . If it is, then  $\exists i_2 \leq j$ .  $(\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!]$  by C.3 and C.5.  
 1004       So, `OP_CONTEXT` runs  $i_2$  times, and then we have the following.  
 1005       SUFFICES: By C.4,  $(\varsigma_{1f} \star \varsigma_{2f}, (v_1, v_2)) \in \mathcal{V}_{k-i_1-i_2}[\![\theta(t_1) \otimes \theta(t_2)]\!]$ .  
 1006        $\langle 2 \rangle 14$ . By C.5 and  $k - i_1 - i_2 \leq k - i_1, k - i_2$ , have  
 1007        $(\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!]$  and  
 1008        $(\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!]$  as needed.  
 1009        $\langle 1 \rangle 8$ . CASE: `TY_LAMBDA`.  
 1010       SUFFICES: By C.2, to show  $(\varsigma, \theta(\delta(\gamma(\mathbf{fun} \ x : t \rightarrow e)))) \in \mathcal{V}_k[\![\theta(t \multimap t')]\!]$ .  
 1011       ASSUME: Arbitrary  $j \leq k$ ,  $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$  such that  $\varsigma \star \varsigma_v$  is defined.  
 1012       SUFFICES:  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(\mathbf{fun} \ x : t \rightarrow e))) v) \in \mathcal{C}_j[\![\theta(t')]\!]$ .  
 1013       SUFFICES:  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(e)))[x/v]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  by C.4.  
 1014        $\langle 2 \rangle 1$ . By induction,  $\forall k. \_k[\![\Theta; \Delta; \Gamma, x : t \vdash e]\!]$ .  
 1015        $\langle 2 \rangle 2$ . Instantiate it  $j - 1, \theta, \delta, \gamma[x \mapsto v], \sigma + \sigma_v$ .  
 1016        $\langle 2 \rangle 3$ . Hence,  $(\varsigma \star \varsigma_v, \theta(\delta(\gamma[x \mapsto v](e)))) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$ .  
 1017        $\langle 2 \rangle 4$ . By 3,  $\theta(\delta(\gamma[x \mapsto v](e))) = \theta(\delta(\gamma(e)))[x/v]$ , we are done.  
 1018        $\langle 1 \rangle 9$ . CASE: `TY_APP`.  
 1019        $\langle 2 \rangle 1$ . By 2c, 3 and C.7, we know there exists the following (for all  $k$ ):  
 1020       1.  $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\![\Gamma_e]\!]$

- 1021 2.  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\Gamma_{e'}]$   
 1022 3.  $\gamma = \gamma_e \cup \gamma_{e'}$   
 1023 4.  $\sigma = \sigma_e + \sigma_{e'}$ .
- 1024  $\langle 2 \rangle 2$ . By induction,  
 1025 1.  $\forall k. {}_k[\Theta; \Delta; \Gamma \vdash e : t' \multimap t]$   
 1026 2.  $\forall k. {}_k[\Theta; \Delta; \Gamma' \vdash e' : t']$ .
- 1027  $\langle 2 \rangle 3$ . Instantiate the first with  $k, \theta, \delta, \gamma_e, \sigma_e$  to conclude  $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\theta(t') \multimap \theta(t)]$ .  
 1028
- 1029  $\langle 2 \rangle 4$ . Instantiate *this* with  $j$  and  $\sigma_{e'}$  and use  $\langle 2 \rangle 1$  to conclude  $\langle \theta(\sigma_e + \sigma_{e'}), \theta(\delta(\gamma_e(e))) \rangle$   
 1030 either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma_f + \sigma_{e'}, e_f \rangle$ .
- 1031  $\langle 2 \rangle 5$ . CASE:  $j$  steps to **err**  
 1032 By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j < k$  steps.
- 1033  $\langle 2 \rangle 6$ . CASE:  $j$  steps to another heap-and-expression.  
 1034 If it is not a value, then OP\_CONTEXT runs  $j$  times and we are done.
- 1035  $\langle 2 \rangle 7$ . If it is, then  $\exists i_e \leq j. (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\theta(t') \multimap \theta(t)] \subseteq \mathcal{V}_{k-j}[\dots]$  by C.3 and C.5.  
 1036 So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1037 SUFFICES: By C.4  $i_e$  times,  $(\varsigma_f \star \varsigma_{e'}, e_f e') \in \mathcal{C}_{k-i_e}[\theta(t')]$ .
- 1038  $\langle 2 \rangle 8$ . By C.5,  $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\Gamma'] \theta \subseteq \mathcal{L}_{k-i_e}[\Gamma'] \theta$ .
- 1039  $\langle 2 \rangle 9$ . So, instantiate the second IH with  $k - i_e, \theta, \delta, \gamma_{e'}, \sigma_{e'}$  to conclude  
 1040  $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}(e')))) \in \mathcal{C}_{k-i_e}[\theta(t')]$ .
- 1041  $\langle 2 \rangle 10$ . Instantiate *this* with  $j - i_e$  and  $\sigma_f$  to conclude  $\langle \theta(\sigma_f + \sigma_{e'}), \theta(\delta(\gamma_{e'}(e')))) \rangle$   
 1042 either takes  $j - i_e$  steps to **err** or  $\langle \sigma_f + \sigma'_f, e'_f \rangle$ .
- 1043  $\langle 2 \rangle 11$ . CASE:  $j - i_e$  steps to **err**  
 1044 By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j - i_e < k - i_e$   
 1045 steps.
- 1046  $\langle 2 \rangle 12$ . CASE:  $j - i_e$  steps to another heap-and-expression.  
 1047 If it is not a value, then OP\_CONTEXT runs  $j - i_e$  times and we are done.
- 1048  $\langle 2 \rangle 13$ . If it is, then  $\exists i_{e'} \leq j - i_e. (\varsigma'_f, v_{e'}) \in \mathcal{V}_{k-i_e-i_{e'}}[\theta(t')]$  by C.3.  
 1049 So, OP\_CONTEXT runs  $i_{e'}$  times, and then we have the following.  
 1050 SUFFICES: By C.4  $i_{e'}$  times,  $(\varsigma_f \star \varsigma'_f, e_f e'_f) \in \mathcal{C}_{k-i_e-i_{e'}}[\theta(t')]$ .
- 1051  $\langle 2 \rangle 14$ . Instantiate  $(\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\theta(t') \multimap \theta(t)]$  with  $k - i_e - i_{e'} \leq k - i_e$  and  
 1052  $(\varsigma_{v'}, v_{e'}) \in \mathcal{V}_{k-i_e-i_{e'}}[\theta(t')]$ , to conclude  $(\varsigma_f \star \varsigma'_f, e_f e'_f) \in \mathcal{C}_{k-i_e-i_{e'}}[\theta(t)]$  as  
 1053 needed.
- 1054  $\langle 1 \rangle 10$ . CASE: TY\_GEN.
- 1055  $\langle 2 \rangle 1$ . By induction,  $\forall k. {}_k[\Theta; fc; \Delta; \Gamma \vdash e : t]$ .
- 1056  $\langle 2 \rangle 2$ . LET:  $f$  be arbitrary;  $\theta' \equiv \theta[fc \mapsto f]$ .

## 23:36 NumLin: Linear Types for Linear Algebra

1057            Instantiate induction hypothesis with  $k - 1, \theta', \delta, \gamma, \sigma$ ,  
 1058            to conclude  $(\varsigma, \theta'(\gamma(\delta(e)))) \in \mathcal{C}_{k-1}[\![\theta'(t)]\!]$  (for all  $f$ , by C.8).

1059         $\langle 2 \rangle 3$ . Instantiate *this* with  $j$  and  $\emptyset$  to conclude  $\langle \theta(\sigma), \theta'(\gamma(\delta(e))) \rangle$   
 1060            either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma', e' \rangle$  (for all  $f$ , by C.8).

1061         $\langle 2 \rangle 4$ . CASE:  $j$  steps to **err**.  
 1062            By OP\_CONTEXT\_ERR, whole expression reduces to **err** in  $j < k - 1$  steps  
 1063            (for  $f = fc$ ).

1064         $\langle 2 \rangle 5$ . CASE:  $j$  steps to another heap-and-expression.  
 1065            If it is not a value, then for  $f = fc$ , OP\_CONTEXT runs  $j$  times and we are  
 1066            done.

1067         $\langle 2 \rangle 6$ . If it is, then  $\exists i_e \leq j. (\varsigma', e') \in \mathcal{V}_{k-1-i_e}[\![\theta'(t)]\!] \subseteq \mathcal{V}_{k-1-j}[\![\dots]\!]$   
 1068            by C.3 and C.5 (for all  $f$ , by C.8).

1069         $\langle 2 \rangle 7$ . So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1070            SUFFICES: By C.4  $i_e$  times,  $(\varsigma', \mathbf{fun} fc \rightarrow e') \in \mathcal{V}_{k-i_e}[\![\theta(\forall fc. t)]\!]$  (for  $f = fc$ ).

1071         $\langle 2 \rangle 8$ . ASSUME: Arbitrary  $f'$ .  
 1072            SUFFICES:  $(\varsigma', e'[fc/f']) \in \mathcal{V}_{k-1-i_e}[\![\theta(t)[fc/f']]\!]$  (for  $f = fc$ ).

1073         $\langle 2 \rangle 9$ . This is true by instantiating  $\langle 2 \rangle 6$  with  $f = f'$ .

1074     $\langle 1 \rangle 11$ . CASE: TY\_SPC.

1075         $\langle 2 \rangle 1$ . By induction,  $\forall k. k[\![\Theta; \Delta; \Gamma \vdash e : \forall fc. t]\!]$ .

1076         $\langle 2 \rangle 2$ . Instantiate with  $k, \theta, \delta, \gamma, \sigma$  to conclude  $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(\forall fc. t)]\!]$ .

1077         $\langle 2 \rangle 3$ . Instantiate *this* with  $j$  and  $\emptyset$  and to conclude  $\langle \theta(\sigma), \theta(\delta(\gamma(e))) \rangle$   
 1078            either takes  $j$  steps to **err** or a heap-and-expression  $\langle \sigma_f, e_f \rangle$ .

1079         $\langle 2 \rangle 4$ . CASE:  $j$  steps to **err**.  
 1080            By OP\_CONTEXT\_ERR, the whole expression reduces to **err** in  $j < k$  steps.

1081         $\langle 2 \rangle 5$ . CASE:  $j$  steps to another heap-and-expression.  
 1082            If it is not a value, then OP\_CONTEXT runs  $j$  times and we are done.

1083         $\langle 2 \rangle 6$ . If it is, then  $\exists i_e \leq j. (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\![\theta(\forall fc. t)]\!] \subseteq \mathcal{V}_{k-j}[\![\dots]\!]$  by C.3 and C.5.  
 1084            So  $e_f \equiv \mathbf{fun} fc \rightarrow v$  for some  $v$ .

1085         $\langle 2 \rangle 7$ . So, OP\_CONTEXT runs  $i_e$  times, and then we have the following.  
 1086            SUFFICES: By C.4  $i_e$  times,  $(\varsigma_f, (\mathbf{fun} fc \rightarrow v)[f]) \in \mathcal{C}_{k-i_e}[\![\theta(t[fc/f])]\!]$ .  
 1087            SUFFICES: By C.4 once more,  $(\varsigma_f, v[fc/f]) \in \mathcal{C}_{k-i_e-1}[\![\theta(t[fc/f])]\!]$ .

1088         $\langle 2 \rangle 8$ . This is true by instantiating  $\langle 2 \rangle 6$  with  $f$  and C.2.

1089     $\langle 1 \rangle 12$ . CASE: TY\_FIX.  
 1090            SUFFICES:  $(\emptyset, \theta(\delta(\mathbf{fix}(g, x : t, e : t')))) \in \mathcal{V}_k[\![\theta(t \multimap t')]\!]$ , by C.2 ( $\sigma = \{\}$ ,  $\gamma = []$ ).  
 1091            ASSUME: Arbitrary  $j \leq k$ ,  $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$  ( $\varsigma = \emptyset$ , so  $\varsigma \star \varsigma_v$  is defined).



1092 LET:  $\tilde{e} \equiv \theta(\delta(e))$ .  
 1093 SUFFICES:  $(\varsigma_v, \mathbf{fix}(g, x : t, \tilde{e} : t') \ v) \in \mathcal{C}_j[\![\theta(t')]\!]$ .  
 1094 SUFFICES:  $(\varsigma_v, \tilde{e} [x/v] [g/\mathbf{fix}(g, x : t, \tilde{e} : t')]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  by C.4.

1095  $\langle 2 \rangle 1$ . By induction,  $\forall k. {}_k[\![\Theta; \Delta, g : t \multimap t'; x : t \vdash e : t']]\!]$ .  
 1096  $\langle 2 \rangle 2$ . Instantiate this with  $j - 1, \delta[g \mapsto \mathbf{fix}(g, x : t, \tilde{e} : t')], \gamma = [x \mapsto v], \sigma_v$ .  
 1097  $\langle 2 \rangle 3$ . We have  $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{V}_{j-1}[\![\theta(t \multimap t')]\!]$ .  
 1098  $\langle 3 \rangle 1$ . Again by induction (over  $k$ ),  $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{C}_{j-1}[\![\theta(t \multimap t')]\!]$ .  
 1099  $\langle 3 \rangle 2$ . Instantiate *this* with  $j = 0$  and  $\emptyset$  and we are done.  
 1100  $\langle 2 \rangle 4$ . We have  $(\varsigma_v, v) \in \mathcal{V}_{j-1}[\![\theta(t)]\!]$  by assumption and C.5.  
 1101  $\langle 2 \rangle 5$ . So we conclude  $(\varsigma_v, \theta(\delta'(\gamma(e)))) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$  as required.

1102  $\langle 1 \rangle 13$ . CASE: TY\_VAR\_LIN.  
 1103 PROVE:  $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .

1104  $\langle 2 \rangle 1$ .  $\Gamma = \{x : t\}$  by assumption of TY\_VAR\_LIN.  
 1105  $\langle 2 \rangle 2$ . SUFFICES:  $(\varsigma, \gamma(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$  by 3 ( $\theta$  and  $\delta$  irrelevant).  
 1106  $\langle 2 \rangle 3$ . By 2c, there exist  $(\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$ , such that  $\varsigma = \varsigma_x$  and  $\gamma = [x \mapsto v_x]$ .  
 1107  $\langle 2 \rangle 4$ . Hence,  $(\varsigma_x, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$ , by C.2.

1108  $\langle 1 \rangle 14$ . CASE: TY\_VAR.  
 1109 PROVE:  $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$ .

1110  $\langle 2 \rangle 1$ .  $x : t \in \Delta$  and  $\Gamma = \emptyset$  by assumption of TY\_VAR.  
 1111  $\langle 2 \rangle 2$ . SUFFICES:  $(\emptyset, \delta(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$  by 3.  
 1112  $\langle 2 \rangle 3$ . By 2b, there exists  $v_x$  such that  $(\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$  ( $\theta$  irrelevant and  $\gamma$  empty).  
 1113  $\langle 2 \rangle 4$ . Hence,  $(\emptyset, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$ , by C.2.

1114  $\langle 1 \rangle 15$ . CASE: TY\_UNIT\_INTRO.  
 1115 True by C.2 and definition of  $\mathcal{V}_k[\![\mathbf{unit}]\!]$ .

1116  $\langle 1 \rangle 16$ . CASE: TY\_BOOL\_TRUE, TY\_BOOL\_FALSE, TY\_INT\_INTRO, TY\_ELT\_INTRO.  
 1117 Similar to TY\_UNIT\_INTRO.

## 1118 D.1 Well-formed types

1119  $\boxed{\Theta \vdash f \text{ Perm}}$  Well-formed fractional permissions

1120  $\frac{fc \in \Theta}{\Theta \vdash fc \text{ Perm}}$  WF\_PERM\_VAR

## 23:38 NumLin: Linear Types for Linear Algebra

1121	$\frac{}{\Theta \vdash 1 \text{ Perm}}$	WF_PERM_ZERO
1122	$\frac{\Theta \vdash f \text{ Perm}}{\Theta \vdash \frac{1}{2}f \text{ Perm}}$	WF_PERM_SUCC
1123	$\boxed{\Theta \vdash t \text{ Type}}$	Well-formed types
1124	$\frac{}{\Theta \vdash \text{unit Type}}$	WF_TYPE_UNIT
1125	$\frac{}{\Theta \vdash \text{bool Type}}$	WF_TYPE_BOOL
1126	$\frac{}{\Theta \vdash \text{int Type}}$	WF_TYPE_INT
1127	$\frac{}{\Theta \vdash \text{elt Type}}$	WF_TYPE_ELT
1128	$\frac{\Theta \vdash f \text{ Perm}}{\Theta \vdash f \text{ arr Type}}$	WF_TYPE_ARRAY
1129	$\frac{\Theta \vdash t \text{ Type}}{\Theta \vdash !t \text{ Type}}$	WF_TYPE_BANG
1130	$\frac{\Theta, fc \vdash t \text{ Type}}{\Theta \vdash \forall fc. t \text{ Type}}$	WF_TYPE_GEN
1131	$\frac{\Theta \vdash t \text{ Type} \quad \Theta \vdash t' \text{ Type}}{\Theta \vdash t \otimes t' \text{ Type}}$	WF_TYPE_PAIR
1132	$\frac{\Theta \vdash t \text{ Type} \quad \Theta \vdash t' \text{ Type}}{\Theta \vdash t \multimap t' \text{ Type}}$	WF_TYPE_LOLLY
1133		

## 1134 E NumLin Grammar

	$m$	::=		matrix expressions
			$M$	matrix variables
			$m + m'$	matrix addition
			$m m'$	matrix multiplication
1135			$(m)$	S
	$f$	::=		fractional permission
			$fc$	variable
			$1$	whole permission
			$\frac{1}{2}f$	

1136	$t$	::=		linear type
		<b>unit</b>		unit
		<b>bool</b>		boolean (true/false)
		<b>int</b>		63-bit integers
		<b>elt</b>		array element
		$f$ <b>arr</b>		arrays
		$f$ <b>mat</b>		matrices
		<b>!t</b>		multiple-use type
		$\forall fc.t$	bind $fc$ in $t$	frac. perm. generalisation
		$t \otimes t'$		pair
		$t \multimap t'$		linear function
		$(t)$	S	parentheses
	$p$	::=		primitive
		<b>not</b>		boolean negation
		$(+)$		integer addition
		$(-)$		integer subtraction
		$(*)$		integer multiplication
		$(/)$		integer division
		$(=)$		integer equality
		$(<)$		integer less-than
		$(+.)$		element addition
		$(-.)$		element subtraction
		$(*.)$		element multiplication
		$(/.)$		element division
		$(=.)$		element equality
		$(<.)$		element less-than
		<b>set</b>		array index assignment
		<b>get</b>		array indexing
		<b>share</b>		share array
		<b>unshare</b>		unshare array
		<b>free</b>		free array
		<b>array</b>		Owl: make array
		<b>copy</b>		Owl: copy array
		<b>sin</b>		Owl: map sine over array
		<b>hypot</b>		Owl: $x_i := \sqrt{x_i^2 + y_i^2}$
		<b>asum</b>		BLAS: $\sum_i  x_i $
		<b>axpy</b>		BLAS: $x := \alpha x + y$
		<b>dot</b>		BLAS: $x \cdot y$
		<b>rotmg</b>		BLAS: see its docs
		<b>scal</b>		BLAS: $x := \alpha x$
		<b>amax</b>		BLAS: $\operatorname{argmax} i : x_i$
		<b>setM</b>		matrix index assignment
		<b>getM</b>		matrix indexing

## 23:40 NumLin: Linear Types for Linear Algebra

		<b>shareM</b>		share matrix
		<b>unshareM</b>		unshare matrix
		<b>freeM</b>		free matrix
		<b>matrix</b>		Owl: make matrix
		<b>copyM</b>		Owl: copy matrix
		<b>copyM_to</b>		Owl: copy matrix onto another
		<b>sizeM</b>		dimension of matrix
		<b>trnsp</b>		transpose matrix
		<b>gemm</b>		BLAS: $C := \alpha A^{T?} B^{T?} + \beta C$
		<b>symm</b>		BLAS: $C := \alpha AB + \beta C$
		<b>posv</b>		BLAS: Cholesky decomp. and solve
		<b>potrs</b>		BLAS: solve with given Cholesky
		<b>syrk</b>		BLAS: $C := \alpha A^{T?} A^{T?} + \beta C$
<b>v</b>	<b>::=</b>			<b>values</b>
		<i>p</i>		primitives
		<i>x</i>		variable
		<b>()</b>		unit introduction
		<b>true</b>		true
		<b>false</b>		false
		<i>k</i>		integer
		<i>l</i>		heap location
<b>1137</b>		<i>el</i>		array element
		<b>Many</b> <i>v</i>		!-introduction
		<b>fun</b> <i>fc</i> $\rightarrow v$		frac. perm. abstraction
		<b>(v, v')</b>		pair introduction
		<b>fun</b> <i>x</i> : <i>t</i> $\rightarrow e$	<b>bind</b> <i>x</i> in <i>e</i>	abstraction
		<b>fix</b> ( <i>g</i> , <i>x</i> : <i>t</i> , <i>e</i> : <i>t'</i> )	<b>bind</b> <i>g</i> $\cup x$ in <i>e</i>	fixpoint
		<b>(v)</b>	<b>S</b>	parentheses
<b>e</b>	<b>::=</b>			<b>expression</b>
		<i>p</i>		primitives
		<i>x</i>		variable
		<b>let</b> <i>x</i> = <i>e</i> in <i>e'</i>	<b>bind</b> <i>x</i> in <i>e'</i>	let binding
		<b>()</b>		unit introduction
		<b>let</b> <b>()</b> = <i>e</i> in <i>e'</i>		unit elimination
		<b>true</b>		true
		<b>false</b>		false
		<b>if</b> <i>e</i> then <i>e</i> <sub>1</sub> else <i>e</i> <sub>2</sub>		if
		<i>k</i>		integer
		<i>l</i>		heap location
		<i>el</i>		array element
		<b>Many</b> <i>e</i>		!-introduction
		<b>let</b> <b>Many</b> <i>x</i> = <i>e</i> in <i>e'</i>		!-elimination
		<b>fun</b> <i>fc</i> $\rightarrow e$		frac. perm. abstraction

		$e[f]$		frac. perm. specialisation
		$(e, e')$		pair introduction
		$\mathbf{let} (a, b) = e \mathbf{in} e'$	$\mathbf{bind} a \cup b \mathbf{in} e'$	pair elimination
		$\mathbf{fun} x : t \rightarrow e$	$\mathbf{bind} x \mathbf{in} e$	abstraction
		$e e'$		application
		$\mathbf{fix} (g, x : t, e : t')$	$\mathbf{bind} g \cup x \mathbf{in} e$	fixpoint
		$(e)$	S	parentheses
$C$	$::=$			evaluation contexts
		$\mathbf{let} x = [-] \mathbf{in} e$	$\mathbf{bind} x \mathbf{in} e$	let binding
		$\mathbf{let} () = [-] \mathbf{in} e$		unit elimination
		$\mathbf{if} [-] \mathbf{then} e_1 \mathbf{else} e_2$		if
		$\mathbf{Many} [-]$		!-introduction
		$\mathbf{let} \mathbf{Many} x = [-] \mathbf{in} e$		!-elimination
		$\mathbf{fun} fc \rightarrow [-]$		frac. perm. abstraction
		$[-][f]$		frac. perm. specialisation
		$([-], e)$		pair introduction
		$(v, [-])$		pair introduction
		$\mathbf{let} (a, b) = [-] \mathbf{in} e$	$\mathbf{bind} a \cup b \mathbf{in} e$	pair elimination
		$[-]e$		application
1138		$v[-]$		application
$\Theta$	$::=$			fractional permission environment
		$\cdot$		
		$\Theta, fc$		
$\Gamma$	$::=$			linear types environment
		$\cdot$		
		$\Gamma, x : t$		
		$\Gamma, \Gamma'$		
$\Delta$	$::=$			intuitionistic types environment
		$\cdot$		
		$\Delta, x : t$		
$\sigma$	$::=$			heap (multiset of triples)
		$\{\}$		empty heap
		$\sigma + \{l \mapsto_f m_{k_1, k_2}\}$		location $l$ points to matrix $m$
$Config$	$::=$			result of small step
		$\langle \sigma, e \rangle$		heap and expression
		<b>err</b>		error

**F** Desugaring NumLin

$$\begin{aligned}
x[e] &\Rightarrow \mathbf{get} \_ x (e) && \text{(similarly for matrices)} \\
x[e_1] := e_2 &\Rightarrow \mathbf{set} \ x (e_1) (e_2) && \text{(similarly for matrices)} \\
\\
pat &::= () \mid x \mid !x \mid \mathbf{Many} \ pat \mid (pat, pat) \\
\mathbf{let} \ !x = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ x = e_1 \mathbf{in} \\
&\quad \mathbf{let} \ \mathbf{Many} \ x = \mathbf{Many} \ (\mathbf{Many} \ x) \mathbf{in} \ e_2 \\
\mathbf{let} \ \mathbf{Many} \langle pat_x \rangle = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ x = x \mathbf{in} \\
&\quad \mathbf{let} \ \langle pat_x \rangle = x \mathbf{in} \ e_2 \\
\mathbf{let} \ (\langle pat_a \rangle, \langle pat_b \rangle) = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ (a, b) = a\_b \mathbf{in} \ \mathbf{let} \ \langle pat_a \rangle = a \mathbf{in} \\
&\quad \mathbf{let} \ \langle pat_b \rangle = b \mathbf{in} \ e_2 \\
\mathbf{fun} \ (\langle pat_x \rangle : t) \rightarrow e &\Rightarrow \mathbf{fun} \ (x : t) \rightarrow \mathbf{let} \ \langle pat_x \rangle = x \mathbf{in} \ e \\
\\
arg &::= \langle pat \rangle : t \mid 'x \text{ (fractional permission variable)} \\
\mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e &\Rightarrow \mathbf{fun} \ \langle arg_1 \rangle \rightarrow .. \mathbf{fun} \ \langle arg_n \rangle \rightarrow e \\
\mathbf{let} \ f \ \langle arg_{1..n} \rangle = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ f = \mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e_1 \mathbf{in} \ e_2 \\
\mathbf{let} \ !f \ \langle arg_{1..n} \rangle = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ f = \mathbf{Many} \ (\mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e_1) \mathbf{in} \ e_2 \\
\mathbf{fixpoint} &\equiv \mathbf{fix} \ (f, x : t, \mathbf{fun} \ \langle arg_{1..n} \rangle \rightarrow e_1 : t') \\
\mathbf{let} \ \mathbf{rec} \ f \ (x : t) \ \langle arg_{1..n} \rangle : t' = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ f = \mathbf{fixpoint} \mathbf{in} \ e_2 \\
\mathbf{let} \ \mathbf{rec} \ !f \ (x : t) \ \langle arg_{1..n} \rangle : t' = e_1 \mathbf{in} \ e_2 &\Rightarrow \mathbf{let} \ \mathbf{Many} \ f = \mathbf{Many} \ \mathbf{fixpoint} \mathbf{in} \ e_2
\end{aligned}$$

1141

**G** Primitives

```

module Arr = Owl.Dense.Ndarray.D
type z = Z
type 'a s = Succ
type 'a arr = A of Arr.arr [@@unboxed]
type 'a mat = M of Arr.arr [@@unboxed]
type 'a bang = Many of 'a [@@unboxed]
module Prim :
sig
  val extract : 'a bang -> 'a
  (** Boolean *)
  val not_ : bool bang -> bool bang
  (** Arithmetic, many omitted for brevity *)
  val addI : int bang -> int bang -> int bang
  val eqI : int bang -> int bang -> bool bang
  (** Arrays *)
  val set : z arr -> int bang -> float bang -> z arr
  val get : 'a arr -> int bang -> 'a arr * float bang
  val share : 'a arr -> 'a s arr * 'a s arr
  val unshare : 'a s arr -> 'a s arr -> 'a arr
  val free : z arr -> unit
  (** Owl *)
  val array : int bang -> z arr
  val copy : 'a arr -> 'a arr * z arr
  val sin : z arr -> z arr
  val hypot : z arr -> 'a arr -> 'a arr * z arr
  (** Level 1 BLAS *)
  val asum : 'a arr -> 'a arr * float bang
  val axpy : float bang -> 'a arr -> z arr -> 'a arr * z arr
  val dot : 'a arr -> 'b arr -> ('a arr * 'b arr) * float bang
  val scal : float bang -> z arr -> z arr
  val amax : 'a arr -> 'a arr * int bang
  (* Matrix, some omitted for brevity *)
  val matrix : int bang -> int bang -> z mat
  val eye : int bang -> z mat
  val copy_mat : 'a mat -> 'a mat * z mat
  val copy_mat_to : 'a mat -> z mat -> 'a mat * z mat
  val size_mat : 'a mat -> 'a mat * (int bang * int bang)
  val transpose : 'a mat -> 'a mat * z mat
  (* Level 3 BLAS/LAPACK *)
  val gemm : float bang -> ('a mat * bool bang) -> ('b mat * bool bang) ->
    float bang -> z mat -> ('a mat * 'b mat) * z mat
  val symm : bool bang -> float bang -> 'a mat -> 'b mat -> float bang ->
    z mat -> ('a mat * 'b mat) * z mat
  val gesv : z mat -> z mat -> z mat * z mat
  val posv : z mat -> z mat -> z mat * z mat
  val potrs : 'a mat -> z mat -> 'a mat * z mat
  val syrk : bool bang -> float bang -> 'a mat -> float bang -> z mat ->
    'a mat * z mat
end

```

1142

**H** Kalman Filters from NumLin and C

```

let kalman sigma h mu r_1 data_1 =
  let h, _p_k_n_p_ = Prim.size_mat h in
  let k, n = _p_k_n_p_ in
  let sigma_hT = Prim.matrix n k in
  let (sigma, h), sigma_hT =
    Prim.gemm (Many 1.) (sigma, Many false) (h, Many true) (Many 0.) sigma_hT in
  let (h, sigma_hT), r_2 =
    Prim.gemm (Many 1.) (h, Many false) (sigma_hT, Many false) (Many 1.) r_1 in
  let k_by_k, x = Prim.posv_flip r_2 sigma_hT in
  let (h, mu), data_2 =
    Prim.gemm (Many 1.) (h, Many false) (mu, Many false) (Many (-1.)) data_1 in
  let (x, data_2), new_mu =
    Prim.gemm (Many 1.) (x, Many false) (data_2, Many false) (Many 1.) mu in
  let x_h = Prim.matrix n n in
  let (x, h), x_h =
    Prim.gemm (Many 1.) (x, Many false) (h, Many false) (Many 0.) x_h in
  let () = Prim.free_mat x in
  let sigma, sigma2 = Prim.copy_mat sigma in
  let (sigma, x_h), new_sigma =
    Prim.symm (Many true) (Many (-1.)) sigma x_h (Many 1.) sigma2 in
  let () = Prim.free_mat x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2))))

```

■ **Figure 16** OCaml code for a Kalman filter, generated (at *compile time*) from the code in Figure 8, passed through `ocamlformat` for presentation.

```

static void kalman( const int n,          const int k,
                  const double *sigma, /* n,n */ const double *h, /* k,n */
                  const double *mu, /* n,1 */ double *r, /* k,k */
                  double *data, /* k,1 */ double **ret_sigma /* n,n */ ) {
  double* n_by_k = (double *) malloc(n * k * sizeof(double));
  cblas_dgemm(RowMajor, NoTrans, Trans, n, k, n, 1., sigma, n, h, n, 0., n_by_k, k);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, k, k, n, 1., h, n, n_by_k, k, 1., r, k);
  LAPACKE_dposv(LAPACK_COL_MAJOR, 'U', k, n, r, k, n_by_k, k);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, k, 1, n, 1., h, n, mu, 1, -1., data, 1);
  cblas_dgemm(RowMajor, NoTrans, NoTrans, n, 1, k, 1., n_by_k, k, data, 1, 1., mu, 1);
  double* n_by_n = (double *) malloc(n * n * sizeof(double));
  cblas_dgemm(RowMajor, NoTrans, NoTrans, n, n, k, 1., n_by_k, k, h, n, 0., n_by_n, n);
  free(n_by_k);
  double* n_by_n2 = (double *) malloc(n * n * sizeof(double));
  cblas_dcopy(n*n, sigma, 1, n_by_n2, 1);
  cblas_dsymm(RowMajor, Right, Upper, n, n, -1., sigma, n, n_by_n, n, 1., n_by_n2, n);
  free(n_by_n);
  *ret_sigma = n_by_n2; }

```

■ **Figure 17** CBLAS/LAPACKE implementation of a Kalman filter. I used C instead of Fortran because it is what Owl uses under the hood and OCaml FFI support for C is better and easier to use than that for Fortran. A distinct ‘measure\_kalman’ function that sandwiches a call to this function with `getrusage` is omitted for brevity.